

vue基础用法&基础原理整理

1. vue基础知识和原理

1.1 初识Vue

框架：新型框架 MVVM：Model-View-ViewModel（虚拟DOM节点、DIFF算法）

- Model：模型，你把数据处理好（比如：Ajax请求到服务器拿数据）；
- View：视图，你把页面布局好；
- ViewModel：剩下的事交给我（Vue），我（Vue）来完成View和Model之间的数据渲染；

把页面结构（节点结构）放到一个对象里面（虚拟节点：VNode），当数据发生改变的时候，会使用DIFF算法，去对比你修改了什么，能够继续使用的节点不修改，尽可能少的修改页面节点（修改数据和修改页面显示是异步的）】。

- 想让Vue工作，就必须创建一个Vue实例，且要传入一个配置对象
- demo容器里的代码依然符合html规范，只不过混入了一些特殊的Vue语法
- demo容器里的代码被称为【Vue模板】
- Vue实例和容器是一一对应的
- 真实开发中只有一个Vue实例，并且会配合着组件一起使用
- {{xxx}}是Vue的语法：插值表达式，{{xxx}}可以读取到data中的所有属性
- 一旦data中的数据发生改变，那么页面中用到该数据的地方也会自动更新(Vue实现的响应式)

初始示例代码

```
1 <!-- 准备好一个容器 -->
2 <div id="demo">
3   <h1>Hello, {{name.toUpperCase()}}, {{address}}</h1>
4 </div>
5 <script type="text/javascript" >
6     Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。
7     //创建Vue实例
8     new Vue({
9         el:'#demo', //el用于指定当前Vue实例为哪个容器服务，值通常为css选择器字符串。
10        data:{ //data中用于存储数据，数据供el所指定的容器去使用，值我们暂时先写成一个
            对象。
11                name:'hello,world',
12                address:'北京'
13            }
14        });
```

```
15 </script>
```

```
16
```

1.2 模板语法

Vue模板语法有2大类:

- 插值语法:

功能: 用于解析标签体内容

写法: {{xxx}}, xxx是js表达式, 且可以直接读取到data中的所有属性

- 指令语法:

功能: 用于解析标签 (包括: 标签属性、标签体内容、绑定事件.....)

举例: v-bind:href="xxx" 或 简写为 :href="xxx", xxx同样要写js表达式, 且可以直接读取到data中的所有属性

代码

```
1 <div id="root">
2   <h1>插值语法</h1>
3   <h3>你好, {{name}}</h3>
4   <hr/>
5   <h1>指令语法</h1>
6   <!-- 这里是展示被Vue指令绑定的属性, 引号内写的是js表达式 -->
7   <a :href="school.url.toUpperCase()" x="hello">点我去{{school.name}}学习1</a>
8   <a :href="school.url" x="hello">点我去{{school.name}}学习2</a>
9 </div>
10 <script>
11   new Vue({
12     el: '#root',
13     data: {
14       name: 'jack',
15       school: {
16         name: '百度',
17         url: '
http://www.baidu.com
        url: '
18       }
19     }
20   })
21 </script>
```

1.3 数据绑定

Vue中有2种数据绑定的方式：

- 单向绑定(v-bind)：数据只能从data流向页面
- 双向绑定(v-model)：数据不仅能从data流向页面，还可以从页面流向data

tips: 1.双向绑定一般都应用在表单类元素上（如：input、select等） 2.v-model:value 可以简写为 v-model，因为v-model默认收集的就是value值

代码

```
1 <div id="root">
2   <!-- 普通写法 单向数据绑定 -->
3     单向数据绑定: <input type="text" v-bind:value="name"><br/>
4     双向数据绑定: <input type="text" v-model:value="name"><br/>
5
6     <!-- 简写 v-model:value 可以简写为 v-model，因为v-model默认收集的就是value值-->
7     单向数据绑定: <input type="text" :value="name"><br/>
8     双向数据绑定: <input type="text" v-model="name"><br/>
9 </div>
10 <script>
11   new Vue({
12     el: '#root',
13     data: {
14       name: 'jack',
15     }
16   })
17 </script>
```

1.4 el与data的两种写法

el有2种写法

- new Vue时候配置el属性
- 先创建Vue实例，随后再通过vm.\$mount('#root')指定el的值

代码

```
1 <script>
2   // 第一种
3   const vm = new Vue({
4     el: '#root',
```

```

5          data:{
6              name:'jack',
7          }
8      })
9
10     // 第二种
11     vm.$mount('#root')
12 </script>

```

data有2种写法

- 对象式
- 函数式

在组件中，data必须使用函数式

代码

```

1 <script>
2     new Vue({
3         el:'#root',
4         // 第一种
5         data:{
6             name:'jack',
7         }
8
9         // 第二种
10        data() {
11            return {
12                name: 'jack'
13            }
14        }
15    })
16 </script>

```

1.5 Vue中的MVVM

- M：模型(Model)：data中的数据
- V：视图(View)：模板代码
- VM：视图模型(ViewModel)：Vue实例

1.6 数据代理

了解数据代理需要js的一些知识：Object.defineProperty(), 属性标志，属性描述符，getter, setter。。。建议学习文章地址：<https://zh.javascript.info/property-descriptors> <https://zh.javascript.info/property-accessors> 这里简单介绍一下：**属性标志**: 对象属性 (properties)，除 **value** 外，还有三个特殊的特性 (attributes)，也就是所谓的“标志”

- writable — 如果为 true，则值可以被修改，否则它是只读的
- enumerable — 如果为 true，则表示是可以遍历的，可以在for..in Object.keys()中遍历出来
- configurable — 如果为 true，则此属性可以被删除，这些特性也可以被修改，否则不可以

Object.getOwnPropertyDescriptor(obj, propertyName)

这个方法是查询有关属性的完整信息 obj是对象， propertyName是属性名

```
1 let user = {
2   name: "John"
3 };
4 let descriptor = Object.getOwnPropertyDescriptor(user, 'name');
5 console.log(descriptor)
6 /* 属性描述符:
7 {
8   "value": "John",
9   "writable": true,
10  "enumerable": true,
11  "configurable": true
12 }
13 */
```

打印结果 **Object.defineProperty(obj, prop, descriptor)** obj: 要定义属性的对象。 prop: 要定义或修改的属性的名称 descriptor: 要定义或修改的属性描述符

```
1 let user = {
2   name: "John"
3 };
4 Object.defineProperty(user, "name", {
5   writable: false
6 });
7 user.name = "Pete";
8 // 打印后还是显示 'John', 无法修改name值
```

其他的属性标志就不演示了，接下来看重点：访问器属性。

访问器属性：

本质上是用于获取和设置值的函数，但从外部代码来看就像常规属性。

访问器属性由 “getter” 和 “setter” 方法表示。在对象字面量中，它们用 get 和 set 表示：

```
1 let obj = {
2   get name() {
3     // 当读取 obj.propName 时，getter 起作用
4   },
5   set name() {
6     // 当执行 obj.name = value 操作时，setter 起作用
7   }
8 }
```

更复杂一点的使用

```
1 let user = {
2   surname: 'gao',
3   name: 'han'
4
5   get fullName() {
6     return this.name + this.surname;
7   }
8 }
9 console.log(user.fullName)
```

从外表看，访问器属性看起来就像一个普通属性。这就是访问器属性的设计思想。我们不以函数的方式 **调用** user.fullName，我们正常 **读取** 它：getter 在幕后运行。

vue的计算属性的底层构造感觉用到了这种思想，我目前还没看过源码，是这样猜想的。截至目前，fullName 只有一个 getter。如果我们尝试赋值操作 user.fullName=，将会出现错误：

```
1 user.fullName = "Test"; // Error (属性只有一个 getter)
```

为 user.fullName 添加一个 setter 来修复它：

```
1 let user = {
2   surname: 'gao',
3   name: 'han'
4
5   get fullName() {
```

```

6         return this.name + ' ' + this.surname;
7     }
8     set fullName(value) {
9         // 这个用到了新语法 结构赋值
10        [this.surname, this.name] = value.split(' ');
11    }
12 }
13 user.fullName = 'Li Hua'
14 console.log(user.name);
15 console.log(user.surname);

```

终于可以介绍数据代理了：

数据代理：通过一个对象代理对另一个对象中属性的操作（读/写）

先来看个案例：

```

1 let obj = {
2     x: 100
3 }
4 let obj2 = {
5     y: 200
6 }

```

这时候提一个需求：我们想要访问 **obj** 中的 **x** 的值，但我们最好不要直接去访问 **obj**，而是想要通过 **obj2** 这个代理对象去访问。

这时候就可以用上 **Object.defineProperty()**，给 **obj2** 添加上访问器属性（也就是getter和setter）

代码

```

1 let obj = {
2     x: 100
3 }
4 let obj2 = {
5     y: 200
6 }
7 Object.defineProperty(obj2, 'x', {
8     get() {
9         return obj.x;
10    },
11    set(value) {
12        obj.x = value;

```

```
13     }  
14  })
```

这就是数据代理，也不难吧 接下来介绍Vue中的数据代理

- Vue中的数据代理：通过vm对象来代理data对象中属性的操作（读/写）
- Vue中数据代理的好处：更加方便的操作data中的数据
- 基本原理：
 - 通过Object.defineProperty()把data对象中所有属性添加到vm上。
 - 为每一个添加到vm上的属性，都指定一个getter/setter。
 - 在getter/setter内部去操作（读/写）data中对应的属性。

我来用一个案例来详细解释这一个过程。

```
1  <!-- 准备好一个容器-->  
2  <div id="root">  
3    <h2>学校名称: {{name}}</h2>  
4    <h2>学校地址: {{address}}</h2>  
5  </div>  
6  <script>  
7      const vm = new Vue({  
8          el: '#root',  
9          data: {  
10             name: 'guang',  
11             address: 'chengdu'  
12         }  
13     })  
14 </script>
```

我们在控制台打印 new 出来的 vm

可以看到，写在配置项中的 data 数据被 绑定到了 vm 对象上，我先来讲结果，是 Vue 将 _data 中的 name, address 数据 代理到 vm 本身上。


```

▼ Vue {_uid: 0, _isVue: true, __v_skip: true, _scope: EffectScope, $options: {...}, ...} ⓘ
  $attrs: (...)
  ▶ $children: []
  ▶ $createElement: f (a, b, c, d)
  ▶ $el: div#root
  ▶ $listeners: (...)
  ▶ $options: {components: {...}, directives: {...}, filters: {...}, el: '#root', _base: f, ...}
  $parent: undefined
  ▶ $refs: {}
  ▶ $root: Vue {_uid: 0, _isVue: true, __v_skip: true, _scope: EffectScope, $options: {...}, ...}
  ▶ $scopedSlots: {}
  ▶ $slots: {}
  $vnode: undefined
  address: (...)
  name: (...)
  __v_skip: true
  ▶ _c: f (a, b, c, d)
  ▶ _data: {__ob__: Observer}
  _directInactive: false
  ▶ _events: {}
  _hasHookEvent: false
  _inactive: null
  _isBeingDestroyed: false
  _isDestroyed: false
  _isMounted: true
  _isVue: true
  ▶ _provided: {}

```

一脸懵逼？先来解释下 `_data` 是啥，`_data` 就是 `vm` 身上的 `_data` 属性，就是下图那个 这个 `_data` 是从哪来的？

```

1  <script>
2
3      const vm = new Vue({
4          el: '#root',
5          // 我们在Vue 初始化的配置项中写了 data 属性。
6          data: {
7              name: 'guang',
8              address: 'chengdu'
9          }
10     })
11 </script>

```

`new Vue` 时，`Vue` 通过一系列处理，将匹配项上的 `data` 数据绑定到了 `_data` 这个属性上，并对这个属性进行了处理（数据劫持），但这个属性就是来源于配置项中的 `data`，我们可以来验证一下。

```

1  <script>
2
3      let data1 = {
4          name: 'guang',
5          address: 'chengdu'

```

```

6      }
7
8      const vm = new Vue({
9        el: '#root',
10       // 我们在Vue 初始化的配置项中写了 data 属性。
11       data: data1
12     })
13   </script>
14

```



打印结果为true，说明两者就是同一个 好了，再回到数据代理上来，将 **vm._data** 中的值，再代理到 vm 本身上来，用vm.name 代替 **vm._data.name**。这就是 Vue 的数据代理 这一切都是通过 **Object.defineProperty()** 来完成的，我来模拟一下这个过程

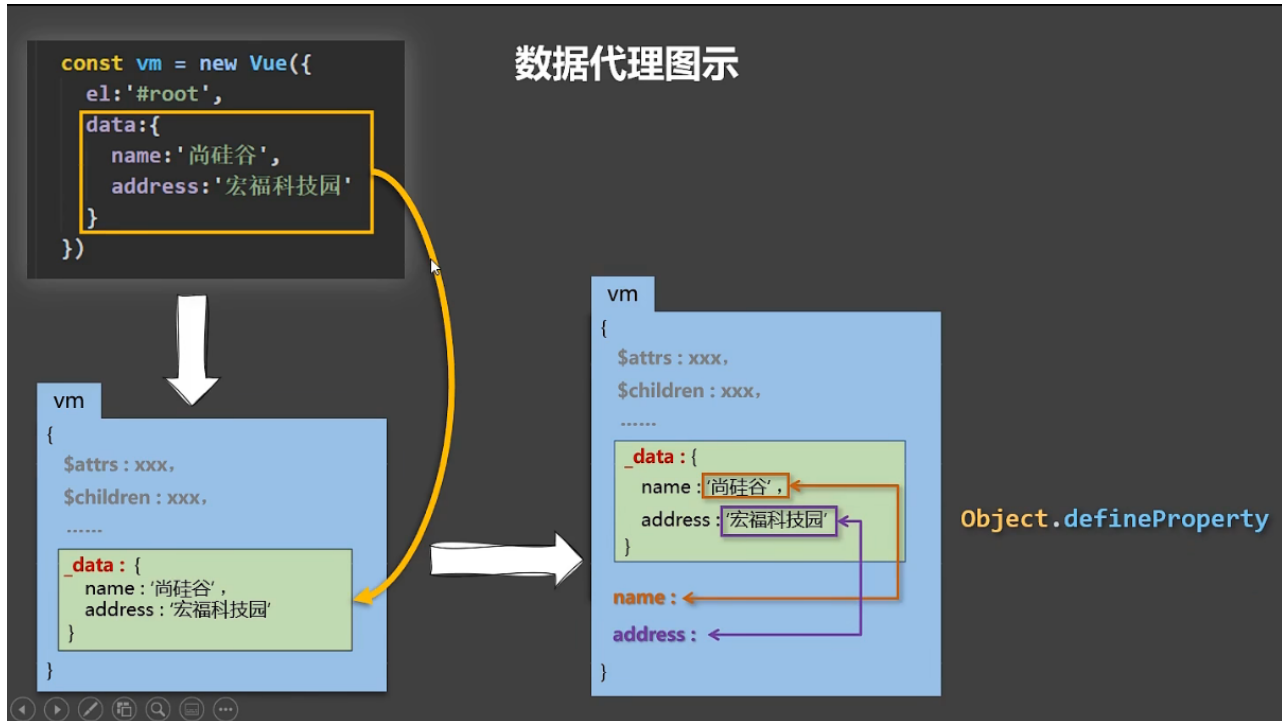
```

1  Object.defineProperty(vm, 'name', {
2    get() {
3      return vm._data.name;
4    },
5    set(value) {
6      vm._data.name = value
7    }
8  })

```

这样有啥意义？明明通过 **vm._data.name** 也可以访问 **name** 的值，为啥费力去这样操作？在插值语法中，**{{ name }}** 取到的值就相当于 **{{ vm.name }}**，不用数据代理的话，在插值语法就要这样去写了。**{{ _data.name }}** 这不符合直觉，怪怪的。vue 这样设计更利于开发者开发，我们在

研究原理会觉得有些复杂（笑~） 来个尚硅谷张天禹老师做的图（非常推荐去看他的课，讲的非常好）



1.7 事件处理

事件的基本使用：

- 使用 `v-on:xxx` 或 `@xxx` 绑定事件，其中 `xxx` 是事件名
- 事件的回调需要配置在 `methods` 对象中，最终会在 `vm` 上
- `methods` 中配置的函数，都是被 `Vue` 所管理的函数，`this` 的指向是 `vm` 或 组件实例对象

```
1 <!-- 准备好一个容器-->
2 <div id="root">
3   <h2>欢迎来到{{name}}学习</h2>
4   <!-- <button v-on:click="showInfo">点我提示信息</button> -->
5   <button @click="showInfo1">点我提示信息1（不传参）</button>
6   <!-- 主动传事件本身 -->
7   <button @click="showInfo2($event,66)">点我提示信息2（传参）</button>
8 </div>
9 <script>
10   const vm = new Vue({
11     el: '#root',
12     data: {
13       name: 'vue',
14     },
15     methods: {
```

```

16      // 如果vue模板没有写event，会自动传 event 给函数
17      showInfo1(event){
18          // console.log(event.target.innerText)
19          // console.log(this) //此处的this是vm
20          alert('同学你好! ')
21      },
22      showInfo2(event,number){
23          console.log(event,number)
24          // console.log(event.target.innerText)
25          // console.log(this) //此处的this是vm
26          alert('同学你好!! ')
27      }
28  }
29  });
30 </script>
31

```

Vue中的事件修饰符

- prevent: 阻止默认事件（常用）
- stop: 阻止事件冒泡（常用）
- once: 事件只触发一次（常用）

```

1  <!-- 准备好一个容器-->
2  <div id="root">
3      <h2>欢迎来到{{name}}学习</h2>
4      <!-- 阻止默认事件（常用） -->
5          <a href="
http://www.baidu.com
        <a href="
6      <!-- 阻止事件冒泡（常用） -->
7      <div class="demo1" @click="showInfo">
8          <button @click.stop="showInfo">点我提示信息</button>
9          <!-- 修饰符可以连续写 -->
10         <!-- <a href="
http://www.atguigu.com
        <!-- <a href="
11     </div>
12     <!-- 事件只触发一次（常用） -->
13     <button @click.once="showInfo">点我提示信息</button>
14 </div>

```

1.8 键盘事件

键盘事件语法糖：@keydown, @keyup

Vue中常用的按键别名：

- 回车 => enter
- 删除 => delete
- 退出 => esc
- 空格 => space
- 换行 => tab (特殊，必须配合keydown去使用)

```

1  <!-- 准备好一个容器-->
2  <div id="root">
3      <h2>欢迎来到{{name}}学习</h2>
4      <input type="text" placeholder="按下回车提示输入" @keydown.enter="showInfo">
5  </div>
6  <script>
7      new Vue({
8          el: '#root',
9          data: {
10             name: '浙江理工大学'
11          },
12          methods: {
13             showInfo(e) {
14                 // console.log(e.key, e.keyCode)
15                 console.log(e.target.value)
16             }
17          },
18      })
19  </script>
20

```

1.9 计算属性

- 定义：要用的属性不存在，要通过已有属性计算得来
- 原理：底层借助了Object.defineProperty方法提供的getter和setter
- get函数什么时候执行？

- (1).初次读取时会执行一次
- (2).当依赖的数据发生改变时会被再次调用
- 优势：与methods实现相比，内部有缓存机制（复用），效率更高，调试方便
- 备注：
 - 计算属性最终会出现在vm上，直接读取使用即可
 - 如果计算属性要被修改，那必须写set函数去响应修改，且set中要引起计算时依赖的数据发生改变

计算属性完整版写法

```

1  <!-- 准备好一个容器-->
2  <div id="root">
3      姓: <input type="text" v-model="firstName">
4      名: <input type="text" v-model="lastName">
5      全名: <span>{{fullName}}</span>
6  </div>
7  <script>
8      const vm = new Vue({
9          el: '#root',
10         data: {
11             firstName: '张',
12             lastName: '三',
13         },
14         computed: {
15             fullName: {
16                 //get有什么作用？当有人读取fullName时，get就会被调用，且返回值就作为fullName
17                 的值
18                 //get什么时候调用？1.初次读取fullName时。2.所依赖的数据发生变化时。
19                 get(){
20                     console.log('get被调用了')
21                     return this.firstName + '-' + this.lastName
22                 },
23                 //set什么时候调用？当fullName被修改时。
24                 // 可以主动在控制台修改fullName来查看情况
25                 set(value){
26                     console.log('set',value)
27                     const arr = value.split('-')
28                     this.firstName = arr[0]
29                     this.lastName = arr[1]
30                 }
31             }
32         }
33     })
  
```

```

30         }
31     }
32 })
33 </script>
34

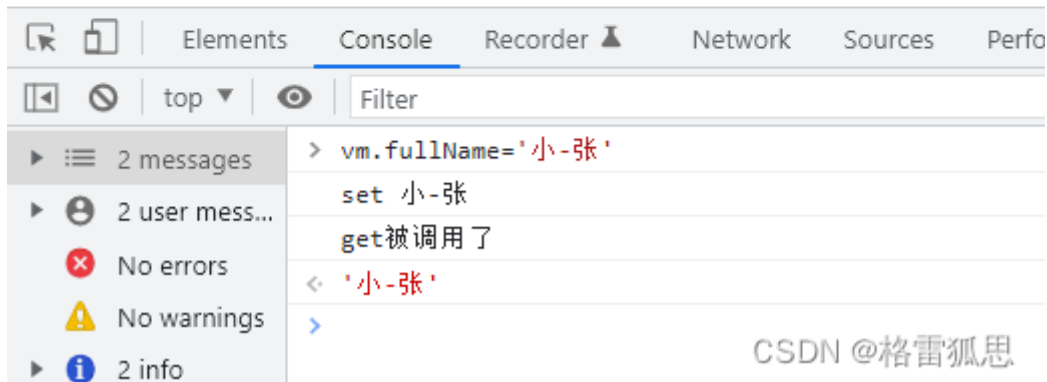
```

姓:

名:

测试:

全名: 小-张



计算属性简写

```

1  <!-- 准备好一个容器-->
2  <div id="root">
3      姓: <input type="text" v-model="firstName">
4      名: <input type="text" v-model="lastName">
5      全名: <span>{{fullName}}</span>
6  </div>
7  <script>
8      const vm = new Vue({
9          el: '#root',
10         data: {
11             firstName: '张',
12             lastName: '三',
13         }
14         computed: {
15             fullName() {

```

```

16         console.log('get被调用了')
17         return this.firstName + '-' + this.lastName
18     }
19 }
20 })
21 </script>
22

```

1.10 监视属性

监视属性watch:

- 当被监视的属性变化时, 回调函数自动调用, 进行相关操作
- 监视的属性必须存在, 才能进行监视
- 监视的两种写法:
 - (1).new Vue时传入watch配置
 - (2).通过vm.\$watch监视

第一种写法

```

1  <!-- 准备好一个容器-->
2  <div id="root">
3      <h2>今天天气很{{ info }}</h2>
4      <button @click="changeWeather">切换天气</button>
5  </div>
6  <script>
7      const vm = new Vue({
8          el:'#root',
9          data:{
10             isHot:true,
11         },
12         computed:{
13             info(){
14                 return this.isHot ? '炎热' : '凉爽'
15             }
16         },
17         methods: {
18             changeWeather(){
19                 this.isHot = !this.isHot
20             }
21         }
22     })
23

```



```

21     },
22     watch:{
23         isHot:{
24             immediate: true, // 初始化时让handler调用一下
25             // handler什么时候调用？当isHot发生改变时。
26             handler(newValue, oldValue){
27                 console.log('isHot被修改了',newValue,oldValue)
28             }
29         }
30     }
31 })
32 </script>
33

```

第二种写法

```

1  <!-- 准备好一个容器-->
2  <div id="root">
3      <h2>今天天气很{{ info }}</h2>
4      <button @click="changeWeather">切换天气</button>
5  </div>
6  <script>
7      const vm = new Vue({
8          el:'#root',
9          data:{
10             isHot:true,
11         },
12         computed:{
13             info(){
14                 return this.isHot ? '炎热' : '凉爽'
15             }
16         },
17         methods: {
18             changeWeather(){
19                 this.isHot = !this.isHot
20             }
21         }
22     })
23

```

```

24     vm.$watch('isHot',{
25         immediate:true, //初始化时让handler调用一下
26         //handler什么时候调用？当isHot发生改变时。
27         handler(newValue,oldValue){
28             console.log('isHot被修改了',newValue,oldValue)
29         }
30     })
31 </script>
32

```

深度监视：

- (1).Vue中的watch默认不监测对象内部值的改变（一层）
- (2).配置deep:true可以监测对象内部值改变（多层）

备注：(1).Vue自身可以监测对象内部值的改变，但Vue提供的watch默认不可以 (2).使用watch时根据数据的具体结构，决定是否采用深度监视

```

1  <!-- 准备好一个容器-->
2  <div id="root">
3      {{numbers.c.d.e}}
4  </div>
5  <script type="text/javascript">
6      Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。
7      const vm = new Vue({
8          el:'#root',
9          data:{
10             numbers:{
11                 c:{
12                     d:{
13                         e:100
14                     }
15                 }
16             }
17         },
18         watch:{
19             //监视多级结构中某个属性的变化
20             /* 'numbers.a':{
21
22                                     handler(){
23                                         console.log('a被改变了')
24                                     }
25             }
26         }
27     })
28
29

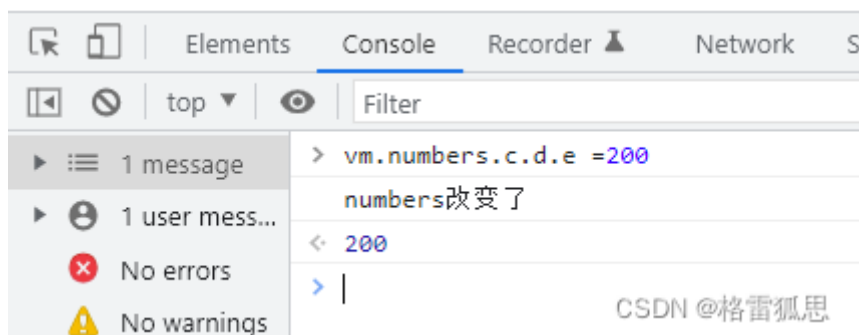
```

```

24         } */
25         //监视多级结构中所有属性的变化
26         numbers:{
27             deep:true,
28             handler(){
29                 console.log('numbers改变了')
30             }
31         }
32     }
33 });
34 </script>
35

```

200



监视属性简写

```

1  <!-- 准备好一个容器-->
2  <div id="root">
3      <h2>今天天气很{{info}}</h2>
4      <button @click="changeWeather">切换天气</button>
5  </div>
6  <script>
7      const vm = new Vue({
8          el:'#root',
9          data:{
10             isHot:true,
11         },

```

```

12     computed:{
13         info(){
14             return this.isHot ? '炎热' : '凉爽'
15         }
16     },
17     methods: {
18         changeWeather(){
19             this.isHot = !this.isHot
20         }
21     },
22     watch:{
23         //简写
24         isHot(newValue, oldValue) {
25             console.log('isHot被修改了', newValue,
26                 oldValue, this)
27         }
28     })
29 </script>
30

```

computed和watch之间的区别：

- computed能完成的功能，watch都可以完成
- watch能完成的功能，computed不一定能完成，例如：watch可以进行异步操作

两个重要的小原则： 1.所被Vue管理的函数，最好写成普通函数，这样this的指向才是vm 或 组件实例对象 2.所有不被Vue所管理的函数（定时器的回调函数、ajax的回调函数等、Promise的回调函数），最好写成箭头函数，这样this的指向才是vm 或 组件实例对象

```

1  <!-- 准备好一个容器-->
2  <div id="root">
3      姓: <input type="text" v-model="firstName"> <br/><br/>
4      名: <input type="text" v-model="lastName"> <br/><br/>
5      全名: <span>{{fullName}}</span> <br/><br/>
6  </div>
7  <script>
8      const vm = new Vue({
9          el:'#root',
10         data:{
11             firstName:'张',

```

```

12         lastName:'三',
13         fullName:'张-三'
14     },
15     watch:{
16         // watch 监视器里可以写 异步函数
17         firstName(val){
18             setTimeout(()=>{
19                 console.log(this)
20                 this.fullName = val + '-' + this.lastName
21             },1000);
22         },
23         lastName(val){
24             this.fullName = this.firstName + '-' + val
25         }
26     }
27 })
28 </script>
29

```

1.11 绑定样式

class样式

写法: :class="xxx" xxx可以是字符串、对象、数。

所以分为三种写法，字符串写法，数组写法，对象写法

字符串写法

字符串写法适用于：类名不确定，要动态获取。

```

1  <style>
2      .normal{
3          background-color: skyblue;
4      }
5  </style>
6  <!-- 准备好一个容器-->
7  <div id="root">
8      <!-- 绑定class样式--字符串写法，适用于：样式的类名不确定，需要动态指定 -->
9      <div class="basic" :class="mood" @click="changeMood">{{name}}</div>
10 </div>
11 <script>

```

```
12      const vm = new Vue({
13        el: '#root',
14        data: {
15          mood: 'normal'
16        }
17      })
18    </script>
19
```

数组写法

数组写法适用于：要绑定多个样式，个数不确定，名字也不确定。

```
1  <style>
2    .atguigu1{
3      background-color: yellowgreen;
4    }
5    .atguigu2{
6      font-size: 30px;
7      text-shadow: 2px 2px 10px red;
8    }
9    .atguigu3{
10     border-radius: 20px;
11   }
12 </style>
13 <!-- 准备好一个容器-->
14 <div id="root">
15   <!-- 绑定class样式--数组写法，适用于：要绑定的样式个数不确定、名字也不确定 -->
16   <div class="basic" :class="classArr">{{name}}</div>
17 </div>
18 <script>
19   const vm = new Vue({
20     el: '#root',
21     data: {
22       classArr: ['atguigu1', 'atguigu2', 'atguigu3']
23     }
24   })
25 </script>
26
```

对象写法

对象写法适用于：要绑定多个样式，个数确定，名字也确定，但不确定用不用。

```
1 <style>
2   .atguigu1{
3     background-color: yellowgreen;
4   }
5   .atguigu2{
6     font-size: 30px;
7     text-shadow: 2px 2px 10px red;
8   }
9 </style>
10 <!-- 准备好一个容器-->
11 <div id="root">
12   <!-- 绑定class样式--对象写法，适用于：要绑定的样式个数确定、名字也确定，但要动态决定用不用 -->
13   <div class="basic" :class="classObj">{{name}}</div>
14 </div>
15 <script>
16   const vm = new Vue({
17     el: '#root',
18     data: {
19       classObj: {
20         atguigu1: false,
21         atguigu2: false,
22       }
23     }
24   })
25 </script>
26
```

style样式

有两种写法，对象写法，数组写法

对象写法

```
1 <!-- 准备好一个容器-->
2 <div id="root">
3   <!-- 绑定style样式--对象写法 -->
```

```
4   <div class="basic" :style="styleObj">{{name}}</div>
5 </div>
6 <script>
7     const vm = new Vue({
8       el: '#root',
9       data: {
10         styleObj: {
11           fontSize: '40px',
12           color: 'red',
13         }
14       }
15     })
16 </script>
17
```

数组写法

```
1  <!-- 准备好一个容器-->
2  <div id="root">
3    <!-- 绑定style样式--数组写法 -->
4    <div class="basic" :style="styleArr">{{name}}</div>
5  </div>
6  <script>
7    const vm = new Vue({
8      el: '#root',
9      data: {
10        styleArr: [
11          {
12            fontSize: '40px',
13            color: 'blue',
14          },
15          {
16            backgroundColor: 'gray'
17          }
18        ]
19      }
20    })
21 </script>
22
```


1.12 条件渲染

v-if

- 写法:

(1).v-if="表达式"

(2).v-else-if="表达式"

(3).v-else="表达式"

- 适用于：切换频率较低的场景
- 特点：不展示的DOM元素直接被移除
- 注意：v-if可以和v-else-if、v-else一起使用，但要求结构不能被“打断”

```
1 <!-- 准备好一个容器-->
2 <div id="root">
3     <!-- 使用v-if做条件渲染 -->
4     <h2 v-if="false">欢迎来到{{name}}</h2>
5     <h2 v-if="1 === 1">欢迎来到{{name}}</h2>
6
7
8     <!-- v-else和v-else-if -->
9     <div v-if="n === 1">Angular</div>
10    <div v-else-if="n === 2">React</div>
11    <div v-else-if="n === 3">Vue</div>
12    <div v-else>哈哈</div>
13
14
15    <!-- v-if与template的配合使用 -->
16    <!-- 就不需要写好多个判断，写一个就行 -->
17    <!-- 这里的思想就像事件代理的使用 -->
18    <template v-if="n === 1">
19        <h2>你好</h2>
20        <h2>尚硅谷</h2>
21        <h2>北京</h2>
22    </template>
23 </div>
24 <script>
25     const vm = new Vue({
26         el: '#root',
27         data: {
```

```

28         styleArr:[
29             {
30                 fontSize: '40px',
31                 color:'blue',
32             },
33             {
34                 backgroundColor:'gray'
35             }
36         ]
37     }
38 })
39 </script>
40

```

v-show

- 写法: v-show="表达式"
- 适用于: 切换频率较高的场景
- 特点: 不展示的DOM元素未被移除, 仅仅是使用样式隐藏掉(display:none)

备注: 使用v-if的时, 元素可能无法获取到, 而使用v-show一定可以获取到 v-if 是实打实地改变 dom元素, v-show 是隐藏或显示dom元素

```

1  <!-- 准备好一个容器-->
2  <div id="root">
3      <!-- 使用v-show做条件渲染 -->
4      <h2 v-show="false">欢迎来到{{name}}</h2>
5      <h2 v-show="1 === 1">欢迎来到{{name}}</h2>
6  </div>
7

```

1.13 列表渲染

v-for指令

- 用于展示列表数据
- 语法: v-for="(item, index) in xxx" :key="yyy"
- 可遍历: 数组、对象、字符串 (用的很少)、指定次数 (用的很少)

```
1 <div id="root">
2   <!-- 遍历数组 -->
3   <h2>人员列表（遍历数组）</h2>
4   <ul>
5     <li v-for="(p,index) of persons" :key="index">
6       {{p.name}}-{{p.age}}
7     </li>
8   </ul>
9   <!-- 遍历对象 -->
10  <h2>汽车信息（遍历对象）</h2>
11  <ul>
12    <li v-for="(value,k) of car" :key="k">
13      {{k}}-{{value}}
14    </li>
15  </ul>
16  <!-- 遍历字符串 -->
17  <h2>测试遍历字符串（用得少）</h2>
18  <ul>
19    <li v-for="(char,index) of str" :key="index">
20      {{char}}-{{index}}
21    </li>
22  </ul>
23  <!-- 遍历指定次数 -->
24  <h2>测试遍历指定次数（用得少）</h2>
25  <ul>
26    <li v-for="(number,index) of 5" :key="index">
27      {{index}}-{{number}}
28    </li>
29  </ul>
30 </div>
31 <script>
32   const vm = new Vue({
33     el:'#root',
34     data: {
35       persons: [
36         { id: '001', name: '张三', age: 18 },
37         { id: '002', name: '李四', age: 19 },
38         { id: '003', name: '王五', age: 20 }
```

```
39         ],
40         car: {
41             name: '奥迪A8',
42             price: '70万',
43             color: '黑色'
44         },
45         str: 'hello'
46     }
47 })
48 </script>
49
```

key的原理

vue中的key有什么作用？（key的内部原理）

了解vue中key的原理需要一些前置知识。

就是vue的虚拟dom，vue会根据 data中的数据生成虚拟dom，如果是第一次生成页面，就将虚拟dom转成真实dom，在页面展示出来。

虚拟dom有啥用？每次vm._data 中的数据更改，都会触发生成新的虚拟dom，新的虚拟dom会跟旧的虚拟dom进行比较，如果有相同的，在生成真实dom时，这部分相同的就不需要重新生成，只需要将两者之间不同的dom转换成真实dom，再与原来的真实dom进行拼接。我的理解是虚拟dom就是起到了一个dom复用的作用，还有避免重复多余的操作，下文有详细解释。

而key有啥用？

key是虚拟dom的标识。

先来点预备的知识：啥是真实 DOM？真实 DOM 和 虚拟 DOM 有啥区别？如何用代码展现真实 DOM 和 虚拟 DOM

真实DOM和其解析流程

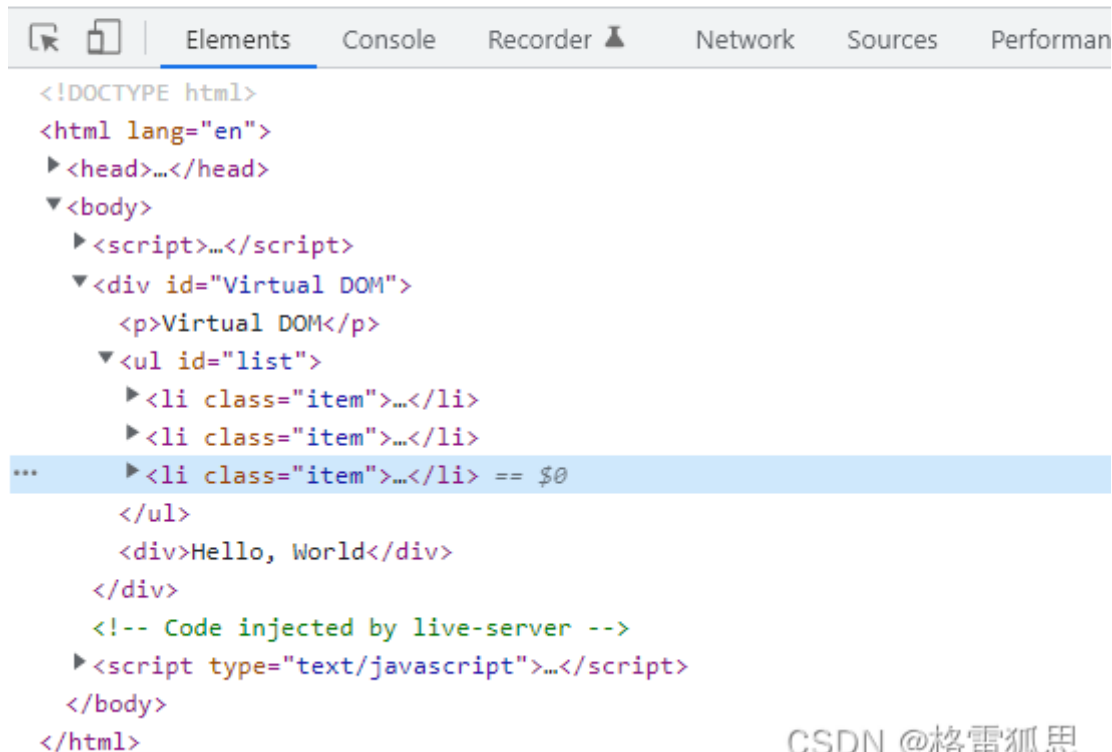
这里参考超级英雄大佬：<https://juejin.cn/post/6844903895467032589>

webkit 渲染引擎工作流程图

Virtual DOM

- Item 1
- Item 2
- Item 3

Hello, World



```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <script>...</script>
    <div id="Virtual DOM">
      <p>Virtual DOM</p>
      <ul id="list">
        <li class="item">...</li>
        <li class="item">...</li>
        ...
        <li class="item">...</li> == $0
      </ul>
      <div>Hello, World</div>
    </div>
    <!-- Code injected by live-server -->
    <script type="text/javascript">...</script>
  </body>
</html>
```

CSDN @格雷狐思

中文版 所有的浏览器渲染引擎工作流程大致分为5步：创建 DOM 树 —> 创建 Style Rules -> 构建 Render 树 —> 布局 Layout —> 绘制 Painting。

- 第一步，构建 DOM 树：当浏览器接收到来自服务器响应的HTML文档后，会遍历文档节点，生成 DOM树。需要注意的是在DOM树生成的过程中有可能会被CSS和JS的加载执行阻塞，渲染阻塞下面会讲到。
- 第二步，生成样式表：用 CSS 分析器，分析 CSS 文件和元素上的 inline 样式，生成页面的样式表；
- 渲染阻塞：当浏览器遇到一个script标签时，DOM构建将暂停，直到脚本加载执行，然后继续构建DOM树。每次去执行Javascript脚本都会严重阻塞DOM树构建，如果JavaScript脚本还操作了CSSOM，而正好这个CSSOM没有下载和构建，那么浏览器甚至会延迟脚本执行和构建DOM，直到这个CSSOM的下载和构建。所以，script标签引入很重要，实际使用时可以遵循下面两个原则：
 - css优先：引入顺序上，css资源先于js资源
 - js后置：js代码放在底部，且js应尽量少影响DOM构建

还有一个小知识：当解析html时，会把新来的元素插入dom树里，同时去查找css，然后把对应的样式规则应用到元素上，查找样式表是按照从右到左的顺序匹配的例如：div p {...}，会先寻找所有p标签并判断它的父标签是否为div之后才决定要不要采用这个样式渲染。所以平时写css尽量用class或者id，不要过度层叠

- 第三步，构建渲染树：通过DOM树和CSS规则我们可以构建渲染树。浏览器会从DOM树根节点开始遍历每个可见节点(注意是可见节点)对每个可见节点，找到其适配的CSS规则并应用。渲染树构建完后，每个节点都是可见节点并且都含有其内容和对应的规则的样式。这也是渲染树和DOM树最大的区别所在。渲染是用于显示，那些不可见的元素就不会在这棵树出现了。除此以外，display none的元素也不会被显示在这棵树里。visibility hidden的元素会出现在这棵树里。
- 第四步，**渲染布局**：布局阶段会从渲染树的根节点开始遍历，然后确定每个节点对象在页面上的确切大小与位置，布局阶段的输出是一个盒子模型，它会精确地捕获每个元素在屏幕内的确切位置与大小。
- 第五步，**渲染树绘制**：在绘制阶段，遍历渲染树，调用渲染器的paint()方法在屏幕上显示其内容。渲染树的绘制工作是由浏览器的UI后端组件完成的。

注意点：

1、DOM 树的构建是文档加载完成开始的？ 构建

DOM 树是一个渐进过程，为达到更好的用户体验，渲染引擎会尽快将内容显示在屏幕上，它不必等到整个

HTML 文档解析完成之后才开始构建

render 树和布局。

2、Render 树是 DOM 树和 CSS 样式表构建完毕后才开始构建的？ 这三个过程在实际进行的时候并不是完全独立的，而是会有交叉，会一边加载，一边解析，以及一边渲染。

3、CSS 的解析注意点？

CSS 的解析是从右往左逆向解析的，嵌套标签越多，解析越慢。

****4、**

JS 操作真实

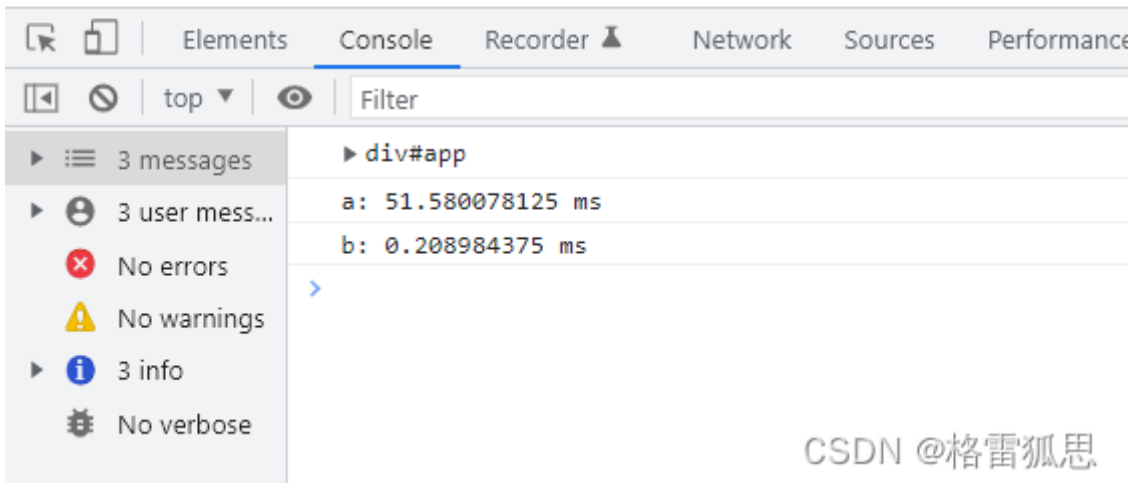
DOM 的代价？**传统DOM结构操作方式对性能的影响很大，原因是频繁操作DOM结构操作会引起页面的重排(reflow)和重绘(repaint)，浏览器不得不频繁地计算布局，重新排列和绘制页面元素，导致浏览器产生巨大的性能开销。直接操作真实

DOM的性能特别差，我们可以来演示一遍。

```
1 <div id="app"></div>
2 <script>
3     // 获取 DIV 元素
4     let box = document.querySelector('#app');
5     console.log(box);
6     // 真实 DOM 操作
7     console.time('a');
8     for (let i = 0; i <= 10000; i++) {
9         box.innerHTML = i;
10    }
```

```
11     console.timeEnd('a');
12     // 虚拟 DOM 操作
13     let num = 0;
14     console.time('b');
15     for (let i = 0; i <= 10000; i++) {
16         num = i;
17     }
18     box.innerHTML = num;
19     console.timeEnd('b');
20 </script>
21
```

10000



CSDN @格雷狐思

从结果中可以看出，操作真实 DOM 的性能是非常差的，所以我们要尽可能的复用，减少 DOM 操作。

虚拟 DOM 的好处

虚拟 DOM 就是为了解决浏览器性能问题而被设计出来的。如前，若一次操作中有 10 次更新 DOM 的动作，虚拟 DOM 不会立即操作 DOM，而是将这 10 次更新的 diff 内容保存到本地一个 JS 对象中，最终将这个 JS 对象一次性 attach 到 DOM 树上，再进行后续操作，避免大量无谓的计算量。所以，用 JS 对象模拟 DOM 节点的好处是，页面的更新可以先全部反映在 JS 对象(虚拟 DOM)上，操作内存中的 JS 对象的速度显然要更快，等更新完成后，再将最终的 JS 对象映射成真实的 DOM，交由浏览器去绘制。

虽然这一个虚拟 DOM 带来的一个优势，但并不是全部。虚拟 DOM 最大的优势在于抽象了原本的渲染过程，实现了跨平台的能力，而不仅仅局限于浏览器的 DOM，可以是安卓和 IOS 的原生组件，可以是近期很火热的小程序，也可以是各种 GUI。

回到最开始的问题，虚拟 DOM 到底是什么，说简单点，就是一个普通的 JavaScript 对象，包含了 tag、props、children 三个属性。

接下来我们手动实现下 虚拟 DOM。分两种实现方式：一种原生 js DOM 操作实现；另一种主流虚拟 DOM 库 (snabbdom、virtual-dom) 的实现 (用 h 函数渲染) (暂时还不理解) **算法实现** ** (1) ** 用 JS 对象模拟 DOM 树：

```
1 <div id="virtual-dom">
2   <p>Virtual DOM</p>
3   <ul id="list">
4     <li class="item">Item 1</li>
5     <li class="item">Item 2</li>
6     <li class="item">Item 3</li>
7   </ul>
8   <div>Hello World</div>
9 </div>
10
```

我们用 JavaScript 对象来表示 DOM 节点，使用对象的属性记录节点的类型、属性、子节点等。

```
1 /**
2  * Element virtual-dom 对象定义
3  * @param {String} tagName - dom 元素名称
4  * @param {Object} props - dom 属性
5  * @param {Array<Element|String>} - 子节点
6  */
7 function Element(tagName, props, children) {
8   this.tagName = tagName;
9   this.props = props;
10  this.children = children;
11  // dom 元素的 key 值，用作唯一标识符
12  if (props.key) {
13    this.key = props.key
14  }
15 }
16 function el(tagName, props, children) {
```



```

17     return new Element(tagName, props, children);
18 }
19

```

构建虚拟的 DOM，用 javascript 对象来表示

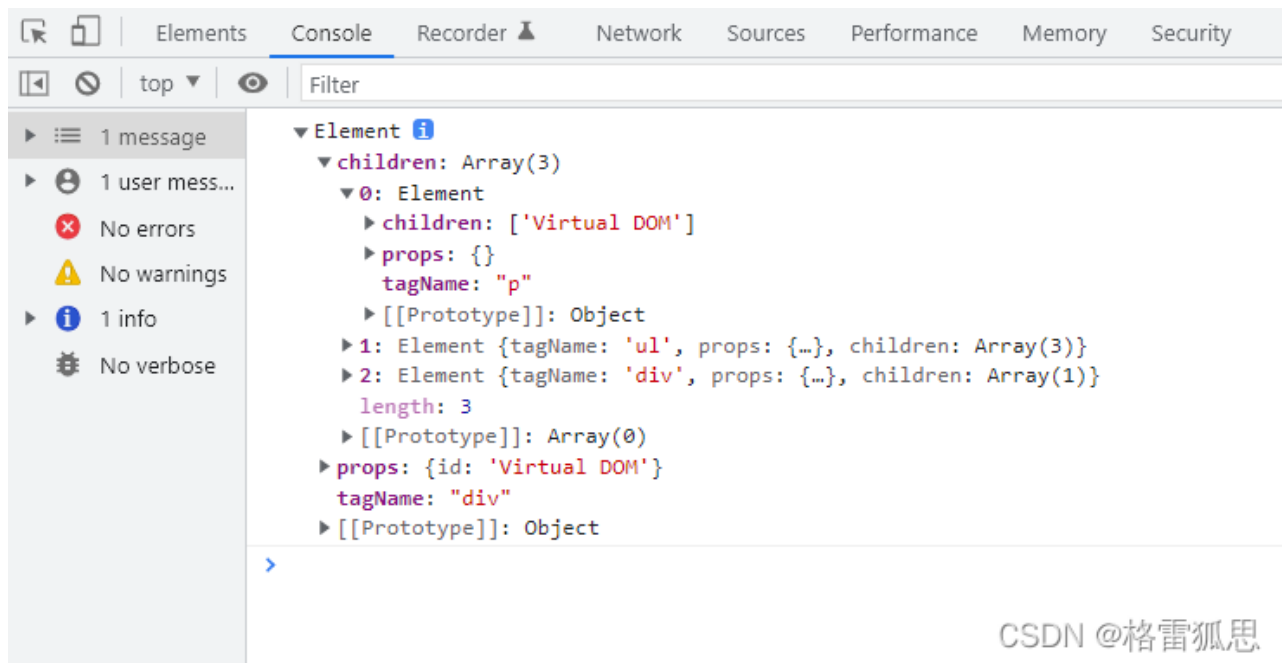
```

1 let ul = el('div', { id: 'Virtual DOM' }, [
2     el('p', {}, ['Virtual DOM']),
3     el('ul', { id: 'list' }, [
4         el('li', { class: 'item' }, ['Item 1']),
5         el('li', { class: 'item' }, ['Item 2']),
6         el('li', { class: 'item' }, ['Item 3'])
7     ]),
8     el('div', {}, ['Hello, World'])
9 ])
10

```

现在 ul 就是我们用 JavaScript 对象表示的 DOM 结构，我们输出查看 ul 对应的数据结构如下：

**** (2) ****将用 js 对象表示的虚拟 DOM 转换成真实 DOM：需要用到 js 原生操作 DOM 的方法。



CSDN @格雷狐思

```

1 /**
2  * render 将virtual-dom 对象渲染为实际 DOM 元素
3  */
4 Element.prototype.render = function () {
5     // 创建节点

```

```

6     let el = document.createElement(this.tagName);
7     let props = this.props;
8     // 设置节点的 DOM 属性
9     for (let propName in props) {
10         let propValue = props[propName];
11         el.setAttribute(propName, propValue)
12     }
13     let children = this.children || []
14     for (let child of children) {
15         let childEl = (child instanceof Element)
16             ? child.render() // 如果子节点也是虚拟 DOM，递归构建 DOM 节点
17             : document.createTextNode(child) // 如果是文本，就构建文本节点
18         el.appendChild(childEl);
19     }
20     return el;
21 }
22

```

我们通过查看以上 render 方法，会根据 tagName 构建一个真正的 DOM 节点，然后设置这个节点的属性，最后递归地把自己的子节点也构建起来。

我们将构建好的 DOM 结构添加到页面 body 上面，如下：

```

1 let ulRoot = ul.render();
2 document.body.appendChild(ulRoot);
3

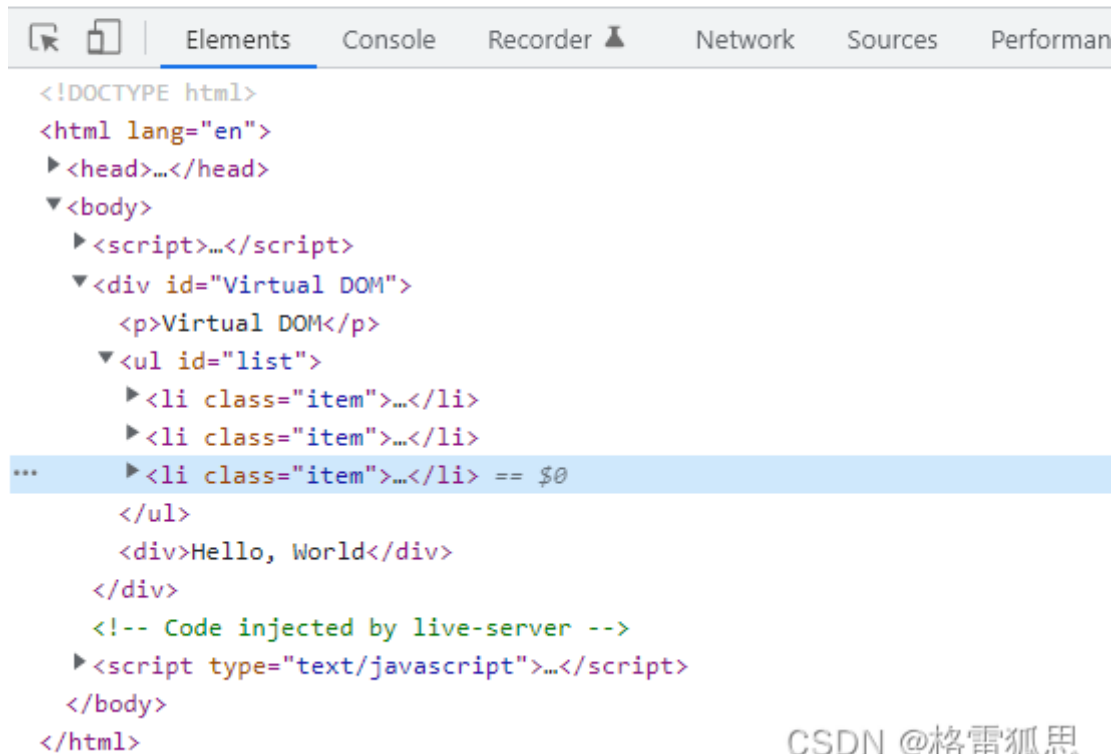
```

这样，页面 body 里面就有真正的 DOM 结构，效果如下图所示：

Virtual DOM

- Item 1
- Item 2
- Item 3

Hello, World



```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <script>...</script>
    <div id="Virtual DOM">
      <p>Virtual DOM</p>
      <ul id="list">
        <li class="item">...</li>
        <li class="item">...</li>
        ...
        <li class="item">...</li> == $0
      </ul>
      <div>Hello, World</div>
    </div>
    <!-- Code injected by live-server -->
    <script type="text/javascript">...</script>
  </body>
</html>
```

CSDN @格雷狐思

我们知道虚拟 DOM 的好处和虚拟 DOM 的实现后就要讲讲 key 的作用了。贴一下上面实现地完整代码

```
1 <script>
2   /**
3     * Element virtual-dom 对象定义
4     * @param {String} tagName - dom 元素名称
5     * @param {Object} props - dom 属性
6     * @param {Array<Element|String>} - 子节点
7     */
8   function Element(tagName, props, children) {
9     this.tagName = tagName;
10    this.props = props;
11    this.children = children;
12    // dom 元素的 key 值，用作唯一标识符
13    if (props.key) {
14      this.key = props.key
```

```
15     }
16 }
17 function el(tagName, props, children) {
18     return new Element(tagName, props, children);
19 }
20 let ul = el('div', { id: 'Virtual DOM' }, [
21     el('p', {}, ['Virtual DOM']),
22     el('ul', { id: 'list' }, [
23         el('li', { class: 'item' }, ['Item 1']),
24         el('li', { class: 'item' }, ['Item 2']),
25         el('li', { class: 'item' }, ['Item 3'])
26     ]),
27     el('div', {}, ['Hello, World'])
28 ])
29 /**
30     * render 将virtual-dom 对象渲染为实际 DOM 元素
31     */
32 Element.prototype.render = function () {
33     // 创建节点
34     let el = document.createElement(this.tagName);
35     let props = this.props;
36     // 设置节点的 DOM 属性
37     for (let propName in props) {
38         let propValue = props[propName];
39         el.setAttribute(propName, propValue)
40     }
41     let children = this.children || []
42     for (let child of children) {
43         let childEl = (child instanceof Element)
44             ? child.render() // 如果子节点也是虚拟 DOM, 递归构建 DOM 节点
45             : document.createTextNode(child) // 如果是文本, 就构建文本节点
46         el.appendChild(childEl);
47     }
48     return el;
49 }
50 let ulRoot = ul.render();
51 document.body.appendChild(ulRoot);
52 console.log(ul);
53 </script>
54
```

虚拟DOM中key的作用

key是虚拟DOM对象的标识，当数据发生变化时，Vue会根据【新数据】生成【新的虚拟DOM】，随后Vue进行【新虚拟DOM】与【旧虚拟DOM】的差异比较，比较规则如下：

- 旧虚拟DOM中找到了与新虚拟DOM相同的key：
 - ①.若虚拟DOM中内容没变, 直接使用之前的真实DOM!
 - ②.若虚拟DOM中内容变了, 则生成新的真实DOM, 随后替换掉页面中之前的真实DOM。
- 旧虚拟DOM中未找到与新虚拟DOM相同的key
 - 创建新的真实DOM, 随后渲染到到页面。

好了，我们知道了最简单的key的原理，如果要继续研究下去就要涉及到vue的核心之一-Diff算法，后面会详细介绍。

用index作为key可能会引发的问题：

若对数据进行：逆序添加、逆序删除等破坏顺序操作：

会产生没有必要的真实DOM更新 ==> 界面效果没问题, 但效率低。

案例

```
1 <!-- 准备好一个容器-->
2 <div id="root">
3   <!-- 遍历数组 -->
4   <h2>人员列表（遍历数组）</h2>
5   <button @click.once="add">添加一个老刘</button>
6   <ul>
7     <li v-for="(p,index) of persons" :key="index">
8       {{p.name}}-{{p.age}}
9       <input type="text">
10    </li>
11  </ul>
12 </div>
13 <script type="text/javascript">
14   Vue.config.productionTip = false
15   new Vue({
16     el: '#root',
17     data: {
18       persons: [
19         { id: '001', name: '张三', age: 18 },
20         { id: '002', name: '李四', age: 19 },
```

```

21             { id: '003', name: '王五', age: 20 }
22         ]
23     },
24     methods: {
25         add() {
26             const p = { id: '004', name: '老刘', age: 40 }
27             this.persons.unshift(p)
28         }
29     },
30 });
31 </script>
32

```

解释： 初始数据 persons: [{ id: '001', name: '张三', age: 18 }, { id: '002', name: '李四', age: 19 }, { id: '003', name: '王五', age: 20 }] **vue根据数据生成虚拟 DOM** 初始虚拟 DOM

```

1 <li key='0'>张三-18<input type="text"></li>
2 <li key='1'>李四-19<input type="text"></li>
3 <li key='2'>王五-20<input type="text"></li>
4

```

将虚拟 DOM 转为 真实 DOM

this.persons.unshift({ id: '004', name: '老刘', age: 40 })

在 persons 数组最前面添加上 { id: '004', name: '老刘', age: 40 }

新数据：

```

persons: [
  { id: '004', name: '老刘', age: 40 },
  { id: '001', name: '张三', age: 18 },
  { id: '002', name: '李四', age: 19 },
  { id: '003', name: '王五', age: 20 }
]

```

vue根据数据生成虚拟 DOM

新虚拟 DOM

人员列表（遍历数组）

添加一个老刘

- 张三-18
- 李四-19
- 王五-20

CSDN @格雷狐思

```
1 <li key='0'>老刘-30<input type="text"></li>
2 <li key='1'>张三-18<input type="text"></li>
3 <li key='3'>李四-19<input type="text"></li>
4 <li key='4'>王五-20<input type="text"></li>
5
```

将虚拟 DOM 转为 真实 DOM

因为老刘被插到第一个，重刷了 key 的值，vue Diff 算法 根据 key 的值 判断 虚拟DOM 全部发生了改变，然后全部重新生成新的 真实 DOM。实际上，张三，李四，王五并没有发生更改，是可以直接复用之前的真实 DOM，而因为 key 的错乱，导致要全部重新生成，造成了性能的浪费。

人员列表（遍历数组）

添加一个老刘

- 老刘-40
- 张三-18
- 李四-19
- 王五-20

CSDN @格雷狐思

来张尚硅谷的图 如果结构中还包含输入类的DOM：会产生错误DOM更新 ==> 界面有问题。这回造成的就不是性能浪费了，会直接导致页面的错误

结论：

- 最好使用每条数据的唯一标识作为key, 比如id、手机号、身份证号、学号等唯一值
- 如果不存在对数据的逆序添加、逆序删除等破坏顺序操作，仅用于渲染列表用于展示，使用index

作为key是没有问题的

来张尚硅谷的图，正经使用 key

1.14 vue 监测data 中的 数据

先来个案例引入一下：

```
1 <!-- 准备好一个容器-->
2 <div id="root">
3   <h2>人员列表</h2>
4   <button @click="updateMei">更新马冬梅的信息</button>
5   <ul>
6     <li v-for="(p,index) of persons" :key="p.id">
7       {{p.name}}-{{p.age}}-{{p.sex}}
8     </li>
9   </ul>
10 </div>
11 <script type="text/javascript">
12   Vue.config.productionTip = false
13   const vm = new Vue({
14     el:'#root',
15     data:{
16       persons:[
17         {id:'001',name:'马冬梅',age:30,sex:'女'},
18         {id:'002',name:'周冬雨',age:31,sex:'女'},
19         {id:'003',name:'周杰伦',age:18,sex:'男'},
20         {id:'004',name:'温兆伦',age:19,sex:'男'}
21       ]
22     },
23     methods: {
24       updateMei(){
25         // this.persons[0].name = '马老师' //奏效
26         // this.persons[0].age = 50 //奏效
27         // this.persons[0].sex = '男' //奏效
28         this.persons[0] = {id:'001',name:'马老师',age:50,sex:'男'} //不奏效
29         // this.persons.splice(0,1,{id:'001',name:'马老师',age:50,sex:'男'})
30       }
31     }
32   })
```



```
33 </script>
```

```
34
```

点击更新马冬梅的信息，马冬梅的数据并没有发生改变。

我们来看看控制台：

控制台上的数据发生了改变，说明，这个更改的数据并没有被 vue 监测到。

所以我们来研究一下 Vue 监测的原理。

我们先研究 Vue 如何监测 对象里的数据

人员列表

更新马冬梅的信息

- 马冬梅-30-女
- 周冬雨-31-女
- 周杰伦-18-男
- 温兆伦-19-男

vue-devtools Detected Vue v2.6.12

1 message

1 user mess...

No errors

No warnings

1 info

No verbose

vm.persons[0]

{id: '001', name: '马老师', age: 50, sex: '男'}

age: 50

id: "001"

name: "马老师"

sex: "男"

[[Prototype]]: Object

CSDN @格雷狐思

人员列表

更新马冬梅的信息

- 马冬梅-30-女
- 周冬雨-31-女
- 周杰伦-18-男
- 温兆伦-19-男

<Root> = \$vm0

<Root>

data

persons: Array[4]

0: Object

age: 30

id: "001"

name: "马冬梅"

sex: "女"

1: Object

2: Object

3: Object

CSDN @格雷狐思

代码

```
1 <!-- 准备好一个容器-->
2 <div id="root">
3   <h2>学校名称: {{name}}</h2>
```

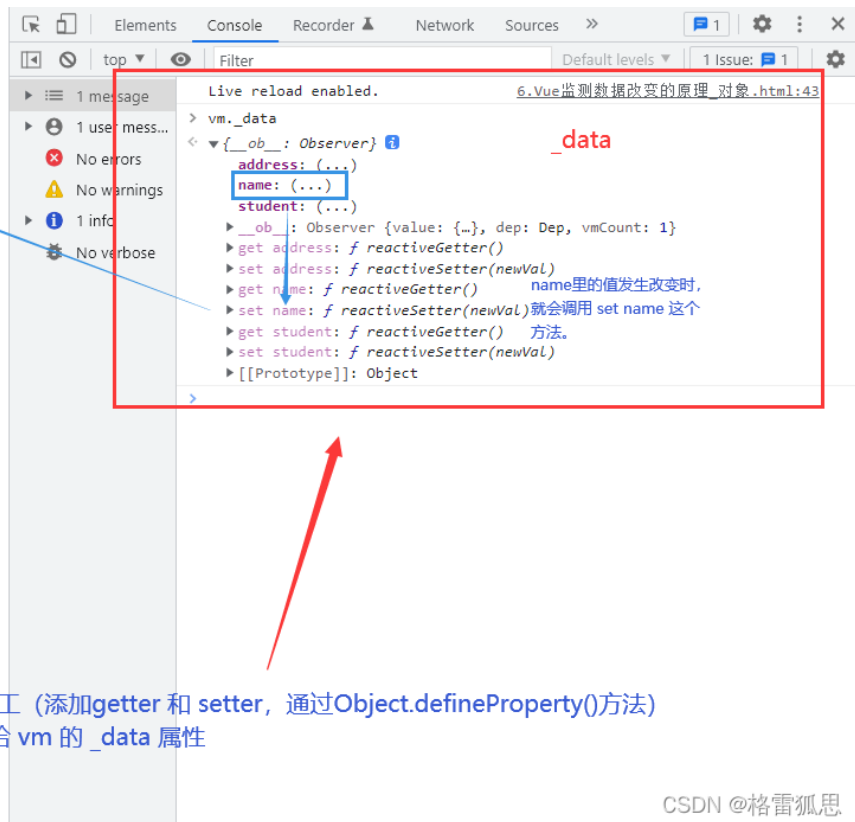
```
4     <h2>学校地址: {{address}}</h2>
5 </div>
6 <script type="text/javascript">
7     Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。
8     const vm = new Vue({
9         el: '#root',
10        data: {
11            name: '浙江师范大学',
12            address: '金华',
13            student: {
14                name: 'tom',
15                age: {
16                    rAge: 40,
17                    sAge: 29,
18                },
19                friends: [
20                    {name: 'jerry', age: 35}
21                ]
22            }
23        }
24    })
25 </script>
26
```

学校名称: 浙江师范大学

学校地址: 金华

调用 set name 这个方法时,
就会去重新渲染 vue 模板,
这就能达到响应式的效果了。

```
data: {  
  name: '浙江师范',  
  address: '金华'  
}
```



讲一下解析模板后面的操作---》调用 set 方法时, 就会去解析模板----->生成新的虚拟 DOM----->新旧DOM 对比 -----> 更新页面 模拟一下 vue 中的 数据监测

```
1 <script type="text/javascript" >  
2   let data = {  
3     name: '尚硅谷',  
4     address: '北京',  
5   }  
6   //创建一个监视的实例对象, 用于监视data中属性的变化  
7   const obs = new Observer(data)  
8   console.log(obs)  
9   //准备一个vm实例对象  
10  let vm = {}  
11  vm._data = data = obs  
12  function Observer(obj){  
13    //汇总对象中所有的属性形成一个数组  
14    const keys = Object.keys(obj)  
15    //遍历  
16    keys.forEach((k) => {  
17      Object.defineProperty(this, k, {  
18        get() {  
19          return obj[k]  
20        },
```

```

21         set(val) {
22             console.log(` ${k}被改了，我要去解析模板，生成虚拟DOM.....我要开始忙了`)
23             obj[k] = val
24         }
25     })
26 })
27 }
28 </script>
29

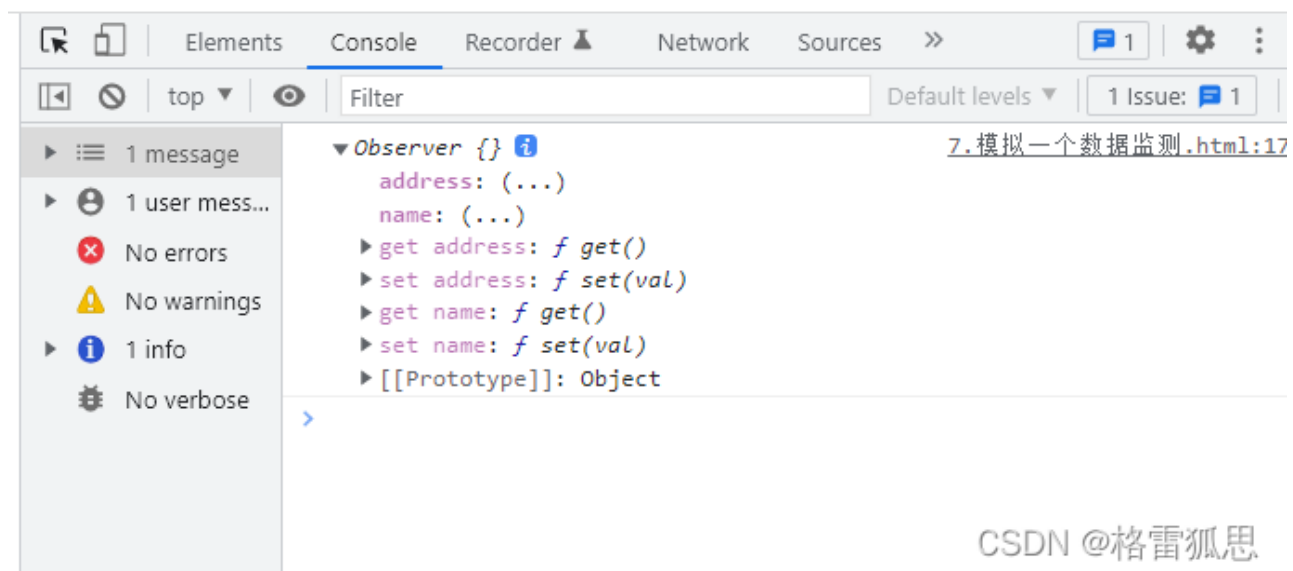
```

Vue.set 的使用

Vue.set(target, propertyName/index, value) 或
vm.\$set(target, propertyName/index, value)

用法：

向响应式对象中添加一个 property，并确保这个新 property 同样是响应式的，且触发视图更新。它必须用于向响应式对象上添加新 property，因为 Vue 无法探测普通的新增 property (比如 vm.myObject.newProperty = 'hi')



CSDN @格雷狐思

代码

```

1  <!-- 准备好一个容器-->
2  <div id="root">
3      <h1>学生信息</h1>
4      <button @click="addSex">添加性别属性，默认值：男</button> <br/>
5  </div>
6  <script type="text/javascript">
7      Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。
8      const vm = new Vue({

```

```

9      el: '#root',
10     data: {
11       student: {
12         name: 'tom',
13         age: 18,
14         hobby: ['抽烟', '喝酒', '烫头'],
15         friends: [
16           {name: 'jerry', age: 35},
17           {name: 'tony', age: 36}
18         ]
19       }
20     },
21     methods: {
22       addSex() {
23         // Vue.set(this.student, 'sex', '男')
24         this.$set(this.student, 'sex', '男')
25       }
26     }
27   })
28 </script>
29

```

Vue.set() 或 vm.\$set 有缺陷:



注意对象不能是 Vue 实例，或者 Vue 实例的根数据对象。

就是 vm 和 _data 看完了 vue 监测对象中的数据，再来看看 vue 如何监测 数组里的数据 先写个代码案例

```

1 <!-- 准备好一个容器-->
2 <div id="root">
3   <h2>爱好</h2>
4   <ul>
5     <li v-for="(h,index) in student.hobby" :key="index">
6       {{h}}
7     </li>
8   </ul>

```

```
9     <h2>朋友们</h2>
10     <ul>
11         <li v-for="(f,index) in student.friends" :key="index">
12             {{f.name}}--{{f.age}}
13         </li>
14     </ul>
15 </div>
16 <script type="text/javascript">
17     Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。
18     const vm = new Vue({
19         el:'#root',
20         data:
21             student:{
22                 name:'tom',
23                 age:{
24                     rAge:40,
25                     sAge:29,
26                 },
27                 hobby:['抽烟','喝酒','烫头'],
28                 friends:[
29                     {name:'jerry',age:35},
30                     {name:'tony',age:36}
31                 ]
32             }
33     },
34     methods: {
35
36     }
37 })
38 </script>
39
```

爱好

- 抽烟
- 喝酒
- 烫头

朋友们

- jerry--35
- tony--36



所以我们通过 `vm._data.student.hobby[0] = 'aaa' // 不奏效` vue 监测在数组那没有 getter 和 setter，所以监测不到数据的更改，也不会引起页面的更新 既然 vue 在对数组无法通过 getter 和 setter 进行数据监视，那 vue 到底如何监视数组数据的变化呢？vue对数组的监测是通过 包装数组上常用的用于修改数组的方法来实现的。vue官网的解释：**总结：**Vue监视数据的原理：

- vue会监视data中所有层次的数据
- 如何监测对象中的数据？

通过setter实现监视，且要在new Vue时就传入要监测的数据。

- 对象中后追加的属性，Vue默认不做响应式处理
- 如需给后添加的属性做响应式，请使用如下API：

`Vue.set(target, propertyName/index, value)` 或
`vm.$set(target, propertyName/index, value)`

- 如何监测数组中的数据？

通过包裹数组更新元素的方法实现，本质就是做了两件事：

- 调用原生对应的方法对数组进行更新
- 重新解析模板，进而更新页面
- 在Vue修改数组中的某个元素一定要用如下方法：
 - 使用这些API: `push()`、`pop()`、`shift()`、`unshift()`、`splice()`、`sort()`、`reverse()`
 - `Vue.set()` 或 `vm.$set()`

特别注意：`Vue.set()` 和 `vm.$set()` 不能给vm 或 vm的根数据对象 添加属性！！

1.15 收集表单数据

若：`<input type="text"/>`，则v-model收集的是value值，用户输入的就是value值。

```
1 <!-- 准备好一个容器-->
2 <div id="root">
3   <form @submit.prevent="demo">
4     账号: <input type="text" v-model.trim="userInfo.account"> <br/><br/>
5     密码: <input type="password" v-model="userInfo.password"> <br/><br/>
```

```

6      年龄: <input type="number" v-model.number="userInfo.age"> <br/><br/>
7      <button>提交</button>
8  </form>
9 </div>
10 <script type="text/javascript">
11     Vue.config.productionTip = false
12     new Vue({
13         el: '#root',
14         data: {
15             userInfo: {
16                 account: '',
17                 password: '',
18                 age: 18,
19             }
20         },
21         methods: {
22             demo() {
23                 console.log(JSON.stringify(this.userInfo))
24             }
25         }
26     })
27 </script>
28

```

若: `<input type="radio"/>`, 则v-model收集的是value值, 且要给标签配置value值。

```

1  <!-- 准备好一个容器-->
2  <div id="root">
3      <form @submit.prevent="demo">
4          性别:
5          男<input type="radio" name="sex" v-model="userInfo.sex" value="male">
6          女<input type="radio" name="sex" v-model="userInfo.sex" value="female">
7      </form>
8  </div>
9  <script type="text/javascript">
10     Vue.config.productionTip = false
11     new Vue({
12         el: '#root',
13         data: {

```



```

14         userInfo:{
15             sex:'female'
16         }
17     },
18     methods: {
19         demo(){
20             console.log(JSON.stringify(this.userInfo))
21         }
22     }
23 })
24 </script>
25

```

若: `<input type="checkbox"/>`

- 没有配置input的value属性, 那么收集的就是checked (勾选 or 未勾选, 是布尔值)
- 配置input的value属性:
 - v-model的初始值是非数组, 那么收集的就是checked (勾选 or 未勾选, 是布尔值)
 - v-model的初始值是数组, 那么收集的的就是value组成的数组

```

1  <!-- 准备好一个容器-->
2  <div id="root">
3      <form @submit.prevent="demo">
4          爱好:
5          学习<input type="checkbox" v-model="userInfo.hobby" value="study">
6          打游戏<input type="checkbox" v-model="userInfo.hobby" value="game">
7          吃饭<input type="checkbox" v-model="userInfo.hobby" value="eat">
8          <br/><br/>
9          所属校区
10         <select v-model="userInfo.city">
11             <option value="">请选择校区</option>
12             <option value="beijing">北京</option>
13             <option value="shanghai">上海</option>
14             <option value="shenzhen">深圳</option>
15             <option value="wuhan">武汉</option>
16         </select>
17         <br/><br/>
18         其他信息:
19         <textarea v-model.lazy="userInfo.other"></textarea> <br/><br/>
20         <input type="checkbox" v-model="userInfo.agree">阅读并接受<a href="

```

<http://www.atguigu.com>

`<input type="checkbox" v-model="userInfo.agree">`阅读并接受[<a href="](#)

21 `<button>`提交`</button>`

22 `</form>`

23 `</div>`

24 `<script type="text/javascript">`

25 `Vue.config.productionTip = false`

26 `new Vue({`

27 `el: '#root',`

28 `data: {`

29 `userInfo: {`

30 `hobby: [],`

31 `city: 'beijing',`

32 `other: '',`

33 `agree: ''`

34 `}`

35 `},`

36 `methods: {`

37 `demo() {`

38 `console.log(JSON.stringify(this.userInfo))`

39 `}`

40 `}`

41 `})`

42 `</script>`

43

账号:

密码:

年龄: 18

性别: 男 ☐ 女 ☒

爱好: 学习 ☒ 打游戏 ☒ 吃饭 ☐

所属校区: 北京

其他信息:

☒ 阅读并接受《用户协议》

v-model绑定的是数组

v-model绑定不是数组, 所以是布尔值

Root

data

- userInfo: Object
 - account: ""
 - age: 18
 - agree: true
 - city: "beijing"
 - hobby: Array[2]
 - 0: "study"
 - 1: "game"
 - other: ""
 - password: ""
 - sex: "female"

CSDN @格雷狐思

备注: v-model的三个修饰符: lazy: 失去焦点再收集数据 number: 输入字符串转为有效的数字 trim: 输入首尾空格过滤

1.16 过滤器 (非重点)

定义: 对要显示的数据进行特定格式化后再显示 (适用于一些简单逻辑的处理)。

语法:

- 注册过滤器: `Vue.filter(name,callback)` 或 `new Vue{filters:{}}`
- 使用过滤器: `{{ xxx | 过滤器名 }}` 或 `v-bind:属性 = "xxx | 过滤器名"`

```
1 <!-- 准备好一个容器-->
2 <div id="root">
3   <h2>显示格式化后的时间</h2>
4   <!-- 计算属性实现 -->
5   <h3>现在是: {{ fmtTime }}</h3>
6   <!-- methods实现 -->
7   <h3>现在是: {{ getFmtTime() }}</h3>
8   <!-- 过滤器实现 -->
9   <h3>现在是: {{time | timeFormater}}</h3>
10  <!-- 过滤器实现 (传参) -->
```

```

11     <h3>现在是: {{time | timeFormater('YYYY_MM_DD') | mySlice}}</h3>
12     <h3 :x="msg | mySlice">尚硅谷</h3>
13 </div>
14 <script type="text/javascript">
15     Vue.config.productionTip = false
16     //全局过滤器
17     Vue.filter('mySlice',function(value){
18         return value.slice(0,4)
19     })
20     new Vue({
21         el:'#root',
22         data:{
23             time:1621561377603, //时间戳
24             msg:'你好，尚硅谷'
25         },
26         computed: {
27             fmtTime(){
28                 return dayjs(this.time).format('YYYY年MM月DD日 HH:mm:ss')
29             }
30         },
31         methods: {
32             getFmtTime(){
33                 return dayjs(this.time).format('YYYY年MM月DD日 HH:mm:ss')
34             }
35         },
36         //局部过滤器
37         filters:{
38             timeFormater(value, str='YYYY年MM月DD日 HH:mm:ss'){
39                 // console.log('@',value)
40                 return dayjs(value).format(str)
41             }
42         }
43     })
44 </script>
45

```

备注： 1.过滤器也可以接收额外参数、多个过滤器也可以串联 2.并没有改变原本的数据, 是产生新的对应的数据

1.17 内置指令

v-text指令：(使用的比较少)

- 1.作用：向其所在的节点中渲染文本内容。
- 2.与插值语法的区别：v-text会替换掉节点中的内容，{{xx}}则不会。

```
1 <!-- 准备好一个容器-->
2 <div id="root">
3     <div>你好, {{name}}</div>
4     <div v-text="name"></div>
5     <div v-text="str"></div>
6 </div>
7 <script type="text/javascript">
8     Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。
9     new Vue({
10         el: '#root',
11         data: {
12             name: '张三',
13             str: '<h3>你好啊! </h3>'
14         }
15     })
16 </script>
17
```

v-html指令：(使用的很少)

- 1.作用：向指定节点中渲染包含html结构的内容。
- 2.与插值语法的区别：
 - v-html会替换掉节点中所有的内容，{{xx}}则不会。
 - v-html可以识别html结构。
- 3.严重注意：v-html有安全性问题！！！！
 - 在网站上动态渲染任意HTML是非常危险的，容易导致XSS攻击。
 - 一定要在可信的内容上使用v-html，永不要用在用户提交的内容上！

```
1 <!-- 准备好一个容器-->
2 <div id="root">
3     <div>你好, {{name}}</div>
4     <div v-html="str"></div>
5     <div v-html="str2"></div>
```

```

6 </div>
7 <script type="text/javascript">
8     Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。
9     new Vue({
10         el: '#root',
11         data: {
12             name: '张三',
13             str: '<h3>你好啊! </h3>',
14             str2: '<a href=javascript:location.href="
http://www.baidu.com?
             str2: '<a href=javascript:location.href="
15         }
16     })
17 </script>
18

```

v-cloak指令 (没有值) :

- 本质是一个特殊属性，Vue实例创建完毕并接管容器后，会删掉v-cloak属性。
- 使用css配合v-cloak可以解决网速慢时页面展示出{{xxx}}的问题。

```

1 <style>
2     [v-cloak]{
3         display:none;
4     }
5 </style>
6 <!-- 准备好一个容器-->
7 <div id="root">
8     <h2 v-cloak>{{name}}</h2>
9 </div>
10 <script type="text/javascript" src="
http://localhost:8080/resource/5s/vue.js
    <script type="text/javascript" src="
11 <script type="text/javascript">
12     console.log(1)
13     Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。
14     new Vue({
15         el: '#root',
16         data: {
17             name: '尚硅谷'
18         }

```

```
19     })
20   </script>
21
```

v-once指令: (用的少)

- v-once所在节点在初次动态渲染后，就视为静态内容了。
- 以后数据的改变不会引起v-once所在结构的更新，可以用于优化性能。

```
1  <!-- 准备好一个容器-->
2  <div id="root">
3    <h2 v-once>初始化的n值是:{{ n }}</h2>
4    <h2>当前的n值是:{{ n }}</h2>
5    <button @click="n++">点我n+1</button>
6  </div>
7  <script type="text/javascript">
8    Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。
9    new Vue({
10      el: '#root',
11      data: {
12        n: 1
13      }
14    })
15  </script>
16
```

v-pre指令: (比较没用)

- 跳过其所在节点的编译过程
- 可利用它跳过：没有使用指令语法、没有使用插值语法的节点，会加快编译

```
1  <!-- 准备好一个容器-->
2  <div id="root">
3    <h2 v-pre>Vue其实很简单</h2>
4    <h2>当前的n值是:{{n}}</h2>
5    <button @click="n++">点我n+1</button>
6  </div>
7  <script type="text/javascript">
8    Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。
9    new Vue({
10      el: '#root',
```

```

11         data:{
12             n:1
13         }
14     })
15 </script>
16

```

1.18 自定义指令

需求1：定义一个v-big指令，和v-text功能类似，但会把绑定的数值放大10倍。

需求2：定义一个v-fbind指令，和v-bind功能类似，但可以让其所绑定的input元素默认获取焦点。

语法：

局部指令：

```

1  directives: {
2    focus: {
3      // 指令的定义
4      inserted: function (el) {
5        el.focus()
6      }
7    }
8  }
9

```

全局指令：

```

1  <script>
2    // 注册一个全局自定义指令 `v-focus`
3    Vue.directive('focus', {
4      // 当被绑定的元素插入到 DOM 中时.....
5      inserted: function (el) {
6        // 聚焦元素
7        el.focus()
8      }
9    })
10 </script>
11

```

配置对象中常用的3个回调：

- bind: 指令与元素成功绑定时调用。
- inserted: 指令所在元素被插入页面时调用。
- update: 指令所在模板结构被重新解析时调用。

理解这三个的调用时机，需要进一步了解 vue 的生命周期，下面会介绍。 定义全局指令

```
1 <!-- 准备好一个容器-->
2 <div id="root">
3     <input type="text" v-fbind:value="n">
4 </div>
5 <script type="text/javascript">
6     Vue.config.productionTip = false
7     //定义全局指令
8     Vue.directive('fbind', {
9         // 指令与元素成功绑定时（一上来）
10        bind(element, binding){
11            element.value = binding.value
12        },
13        // 指令所在元素被插入页面时
14        inserted(element, binding){
15            element.focus()
16        },
17        // 指令所在的模板被重新解析时
18        update(element, binding){
19            element.value = binding.value
20        }
21    })
22
23    new Vue({
24        el: '#root',
25        data:{
26            name: '尚硅谷',
27            n: 1
28        }
29    })
30 </script>
31
```

局部指令：

```

1  new Vue({
2    el: '#root',
3    data: {
4      name: '尚硅谷',
5      n: 1
6    },
7    directives: {
8      // big函数何时会被调用? 1.指令与元素成功绑定时（一上来）。2.指令所在的模板被重新解析
      时。
9      /* 'big-number'(element, binding){
10                                     // console.log('big')
11                                     element.innerText = binding.value * 10
12                                     }, */
13      big (element, binding){
14        console.log('big', this) //注意此处的this是window
15        // console.log('big')
16        element.innerText = binding.value * 10
17      },
18      fbind: {
19        //指令与元素成功绑定时（一上来）
20        bind (element, binding){
21          element.value = binding.value
22        },
23        //指令所在元素被插入页面时
24        inserted (element, binding){
25          element.focus()
26        },
27        //指令所在的模板被重新解析时
28        update (element, binding){
29          element.value = binding.value
30        }
31      }
32    }
33  })
34

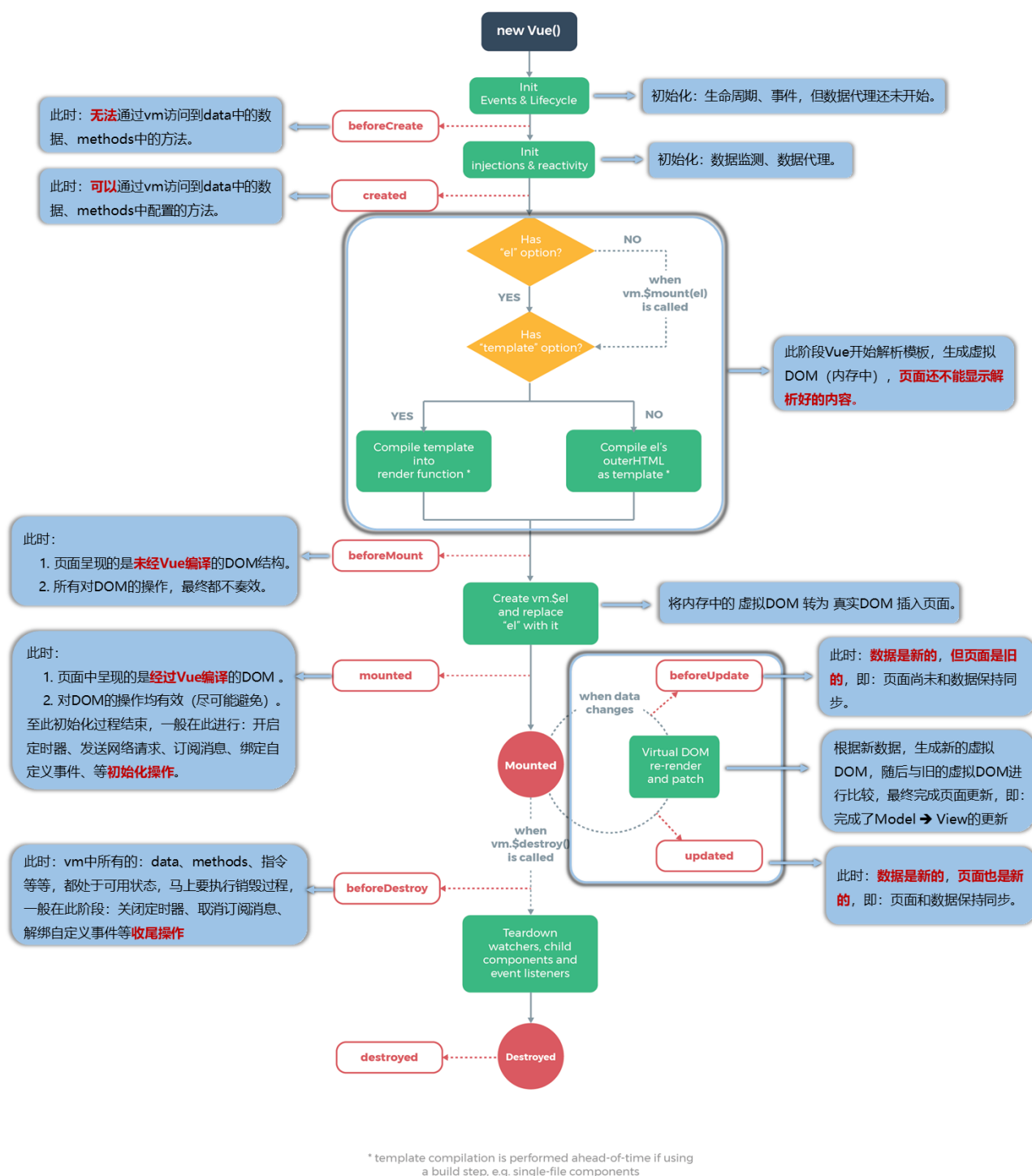
```

1.19 生命周期

简介生命周期

Vue 实例有一个完整的生命周期，也就是从new Vue()、初始化事件(.once事件)和生命周期、编译模版、挂载Dom -> 渲染、更新 -> 渲染、卸载 等一系列过程，称这是Vue的生命周期。

先来一张尚硅谷的图：



CSDN @格雷狐思

1. beforeCreate（创建前）：数据监测(getter和setter)和初始化事件还未开始，此时 data 的响应式追踪、event/watcher 都还没有被设置，也就是说不能访问到data、computed、watch、methods上的方法和数据。

2. **created** (创建后) : 实例创建完成, 实例上配置的 options 包括 data、computed、watch、methods 等都配置完成, 但是此时渲染得节点还未挂载到 DOM, 所以不能访问到 `$el` 属性。
3. **beforeMount** (挂载前) : 在挂载开始之前被调用, 相关的render函数首次被调用。此阶段Vue开始解析模板, 生成虚拟DOM存在内存中, 还没有把虚拟DOM转换成真实DOM, 插入页面中。所以网页不能显示解析好的内容。
4. **mounted** (挂载后) : 在`el`被新创建的 `vm.$el` (就是真实DOM的拷贝) 替换, 并挂载到实例上去之后调用 (将内存中的虚拟DOM转为真实DOM, 真实DOM插入页面)。此时页面中呈现的是经过Vue编译的DOM, 这时在这个钩子函数中对DOM的操作可以有效, 但要尽量避免。一般在这个阶段进行: 开启定时器, 发送网络请求, 订阅消息, 绑定自定义事件等等
5. **beforeUpdate** (更新前) : 响应式数据更新时调用, 此时虽然响应式数据更新了, 但是对应的真实 DOM 还没有被渲染 (数据是新的, 但页面是旧的, 页面和数据没保持同步呢)。
6. **updated** (更新后) : 在由于数据更改导致的虚拟DOM重新渲染和打补丁之后调用。此时 DOM 已经根据响应式数据的变化更新了。调用时, 组件 DOM已经更新, 所以可以执行依赖于DOM的操作。然而在大多数情况下, 应该避免在此期间更改状态, 因为这可能会导致更新无限循环。该钩子在服务器端渲染期间不被调用。
7. **beforeDestroy** (销毁前) : 实例销毁之前调用。这一步, 实例仍然完全可用, `this` 仍能获取到实例。在这个阶段一般进行关闭定时器, 取消订阅消息, 解绑自定义事件。
8. **destroyed** (销毁后) : 实例销毁后调用, 调用后, Vue 实例指示的所有东西都会解绑定, 所有的事件监听器会被移除, 所有的子实例也会被销毁。该钩子在服务端渲染期间不被调用。

来讲一下图中间大框框的内容 先判断有没有 `el` 这个配置项, 没有就调用 `vm.$mount(el)`, 如果两个都没有就一直卡着, 显示的界面就是最原始的容器的界面。有 `el` 这个配置项, 就进行判断有没有 `template` 这个配置项, 没有 `template` 就将 `el` 绑定的容器编译为 vue 模板, 来个对比图。

没编译前的: 编译后: 这个 `template` 有啥用咧? **第一种情况, 有 `template`:** 如果 `el` 绑定的容器没有任何内容, 就一个空壳子, 但在 Vue 实例中写了 `template`, 就会编译解析这个 `template` 里的内容, 生成虚拟 DOM, 最后将 虚拟 DOM 转为 真实 DOM 插入页面 (其实就可以理解为 `template` 替代了 `el` 绑定的容器的内容)。**第二种情况, 没有 `template`:** 没有 `template`, 就编译解析 `el` 绑定的容器, 生成虚拟 DOM, 后面就顺着生命周期执行下去。

1.20 非单文件组件

基本使用

Vue中使用组件的三大步骤:

- 定义组件(创建组件)
- 注册组件
- 使用组件(写组件标签)

定义组件

使用Vue.extend(options)创建，其中options和new Vue(options)时传入的那个options几乎一样，但也有点区别；

区别如下：

- el不要写，为什么？——最终所有的组件都要经过一个vm的管理，由vm中的el决定服务哪个容器。
- data必须写成函数，为什么？——避免组件被复用时，数据存在引用关系。

讲解一下面试小问题：data必须写成函数：

这是js底层设计的原因：举个例子

对象形式

```
1 let data = {
2   a: 99,
3   b: 100
4 }
5 let x = data;
6 let y = data;
7 // x 和 y 引用的都是同一个对象，修改 x 的值， y 的值也会改变
8 x.a = 66;
9 console.log(x); // a:66 b:100
10 console.log(y); // a:66 b:100
11
```

函数形式

```
1 function data() {
2   return {
3     a: 99,
4     b: 100
5   }
6 }
7 let x = data();
8 let y = data();
9 console.log(x === y); // false
10 // 我的理解是函数每调用一次就创建一个新的对象返回给他们
11
```

备注：使用template可以配置组件结构。 创建一个组件案例：Vue.extend() 创建

```
1 <script type="text/javascript">
2   Vue.config.productionTip = false
3   //第一步：创建school组件
4   const school = Vue.extend({
5     template:`
6
7               <div class="demo">
8
9                 <h2>学校名称: {{schoolName}}</h2>
10                <h2>学校地址: {{address}}</h2>
11                <button @click="showName">点我提示学校名
12            </button>
13        </div>
14        `,
15        // el:'#root', //组件定义时，一定不要写el配置项，因为最终所有的组件都要被一个vm管理，
16        // 由vm决定服务于哪个容器。
17        data(){
18            return {
19                schoolName:'尚硅谷',
20                address:'北京昌平'
21            }
22        },
23        methods: {
24            showName(){
25                alert(this.schoolName)
26            }
27        },
28    })
29    //第一步：创建student组件
30    const student = Vue.extend({
31        template:`
32
33                <div>
34
35                    <h2>学生姓名: {{studentName}}</h2>
36                    <h2>学生年龄: {{age}}</h2>
37                </div>
38            `,
39        data(){
40            return {
41                studentName:'张三',
42                age:18
43            }
44        }
45    })
```

```

39     })
40     //第一步：创建hello组件
41     const hello = Vue.extend({
42         template:`
43
44             <div>
45
46                 <h2>你好啊！ {{name}}</h2>
47
48             </div>
49
50         `,
51         data(){
52             return {
53                 name: 'Tom'
54             }
55         }
56     })
57 </script>
58
59

```

注册组件

- 局部注册：靠new Vue的时候传入components选项
- 全局注册：靠Vue.component('组件名',组件)

局部注册

```

1 <script>
2     //创建vm
3     new Vue({
4         el: '#root',
5         data: {
6             msg: '你好啊！ '
7         },
8         //第二步：注册组件（局部注册）
9         components: {
10             school: school,
11             student: student
12             // ES6简写形式
13             // school,
14             // student
15         }
16     })
17 </script>

```

全局注册

```

1 <script>
2     //第二步：全局注册组件
3     Vue.component('hello', hello)
4 </script>
5

```

写组件标签

```

1 <!-- 准备好一个容器-->
2 <div id="root">
3     <hello></hello>
4     <hr>
5     <h1>{{msg}}</h1>
6     <hr>
7     <!-- 第三步：编写组件标签 -->
8     <school></school>
9     <hr>
10    <!-- 第三步：编写组件标签 -->
11    <student></student>
12 </div>
13

```

几个注意点：

关于组件名：

一个单词组成：

- 第一种写法(首字母小写)：school
- 第二种写法(首字母大写)：School

多个单词组成：

- 第一种写法(kebab-case命名)：my-school
- 第二种写法(CamelCase命名)：MySchool (需要Vue脚手架支持)

备注：(1).组件名尽可能回避HTML中已有的元素名称，例如：h2、H2都不行。(2).可以使用name配置项指定组件在开发者工具中呈现的名字。关于组件标签: 第一种写法：<school></school> 第二种写法：<school/> 备注：不用使用脚手架时，<school/>会导致后续组件不能

渲染。一个简写方式： `const school = Vue.extend(options)` 可简写为： `const school = options`

组件的嵌套

比较简单，直接展示代码：

```
1  <!-- 准备好一个容器-->
2  <div id="root">
3  </div>
4  <script type="text/javascript">
5      Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。
6      //定义student组件
7      const student = Vue.extend({
8          name:'student',
9          template:`
10
11                      <div>
12                          <h2>学生姓名: {{name}}</h2>
13                          <h2>学生年龄: {{age}}</h2>
14                      </div>
15          `,
16          data(){
17              return {
18                  name:'尚硅谷',
19                  age:18
20              }
21          })
22      //定义school组件
23      const school = Vue.extend({
24          name:'school',
25          template:`
26
27                      <div>
28                          <h2>学校名称: {{name}}</h2>
29                          <h2>学校地址: {{address}}</h2>
30                          <student></student>
31                      </div>
32          `,
33          data(){
```

```
33         return {
34             name: '尚硅谷',
35             address: '北京'
36         }
37     },
38     // 注册组件（局部）
39     components: {
40         student
41     }
42 })
43 //定义hello组件
44 const hello = Vue.extend({
45     template: `<h1>{{msg}}</h1>`,
46     data(){
47         return {
48             msg: '欢迎来到尚硅谷学习！'
49         }
50     }
51 })
52 //定义app组件
53 const app = Vue.extend({
54     template: `
55         <div>
56             <hello></hello>
57             <school></school>
58         </div>
59     `,
60     components: {
61         school,
62         hello
63     }
64 })
65 //创建vm
66 new Vue({
67     template: '<app></app>',
68     el: '#root',
69     //注册组件（局部）
70     components: {app}
71 })
72 </script>
```

VueComponent

- school组件本质是一个名为VueComponent的构造函数，且不是程序员定义的，是Vue.extend生成的。
- 我们只需要写<school/>或<school> </school>，Vue解析时会帮我们创建school组件的实例对象，即Vue帮我们执行的：new VueComponent(options)。
- 特别注意：每次调用Vue.extend，返回的都是一个全新的VueComponent!!!! (这个VueComponent可不是实例对象)
- 关于this指向：
 - 组件配置中：data函数、methods中的函数、watch中的函数、computed中的函数 它们的this均是【VueComponent实例对象】。
 - new Vue(options)配置中：data函数、methods中的函数、watch中的函数、computed中的函数 它们的this均是【Vue实例对象】。
- VueComponent的实例对象，以后简称vc（也可称之为：组件实例对象）。Vue的实例对象，以后简称vm。

Vue 在哪管理 VueComponent

一个重要的内置关系

- 一个重要的内置关系：VueComponent.prototype.**proto** === Vue.prototype
- 为什么要有这个关系：让组件实例对象（vc）可以访问到 Vue原型上的属性、方法。

1.21 单文件组件

单文件组件就是将一个组件的代码写在 .vue 这种格式的文件中，webpack 会将 .vue 文件解析成 html,css,js这些形式。

来做个单文件组件的案例：

School.vue

```

1  <template>
2  <div class="demo">
3      <h2>学校名称：{{name}}</h2>
4      <h2>学校地址：{{address}}</h2>
5      <button @click="showName">点我提示学校名</button>
6  </div>

```

```

7 </template>
8 <script>
9     export default {
10         name: 'School',
11         data(){
12             return {
13                 name: '尚硅谷',
14                 address: '北京昌平'
15             }
16         },
17         methods: {
18             showName(){
19                 alert(this.name)
20             }
21         },
22     }
23 </script>
24 <style>
25     .demo{
26         background-color: orange;
27     }
28 </style>
29

```

Student.vue

```

1 <template>
2   <div>
3
4       <h2>学生姓名: {{name}}</h2>
5       <h2>学生年龄: {{age}}</h2>
6   </div>
7 </template>
8 <script>
9     export default {
10         name: 'Student',
11         data(){
12             return {
13                 name: '张三',
14                 age: 18
15             }
16         }
17     }
18 </script>
19

```

```

14         }
15     }
16 }
17 </script>
18

```

App.vue

用来汇总所有的组件(大总管)

```

1  <template>
2    <div>
3        <School></School>
4        <Student></Student>
5    </div>
6 </template>
7 <script>
8     //引入组件
9     import School from './School.vue'
10    import Student from './Student.vue'
11    export default {
12        name: 'App',
13        components:{
14            School,
15            Student
16        }
17    }
18 </script>
19

```

main.js

在这个文件里面创建 vue 实例

```

1  import App from './App.vue'
2  new Vue({
3    el: '#root',
4    template: `<App></App>`,
5    components: {App},
6  })
7

```

index.html

在这写 vue 要绑定的容器

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8" />
5     <title>练习一下单文件组件的语法</title>
6   </head>
7   <body>
8     <!-- 准备一个容器 -->
9     <div id="root"></div>
10    <script type="text/javascript" src="../js/vue.js"></script>
11    <script type="text/javascript" src="../main.js"></script>
12  </body>
13 </html>
14
```

2. vue脚手架，自定义事件，插槽等复杂内容

2.1 脚手架

使用前置： 第一步(没有安装过的执行)：全局安装 @vue/cli npm install -g @vue/cli 第二步：切换到要创建项目的目录，然后使用命令创建项目 vue create xxxxx 第三步：启动项目 npm run serve

脚手架文件结构

```
1 |— node_modules
2 |— public
3 |   |— favicon.ico: 页签图标
4 |   └─ index.html: 主页面
5 |— src
6 |   |— assets: 存放静态资源
7 |   |   └─ logo.png
8 |   └─ component: 存放组件
9 |       └─ HelloWorld.vue
```

```
10 | |— App.vue: 汇总所有组件
11 | |— main.js: 入口文件
12 |— .gitignore: git版本管制忽略的配置
13 |— babel.config.js: babel的配置文件
14 |— package.json: 应用包配置文件
15 |— README.md: 应用描述文件
16 |— package-lock.json: 包版本控制文件
17
```

脚手架demo

components:

就直接把单文件组件的 School.vue 和 Student.vue 两个文件直接拿来用，不需要修改。

App.vue:

引入这两个组件，注册一下这两个组件，再使用。

```
1 <template>
2   <div id="app">
3     
4     <Student></Student>
5     <School></School>
6   </div>
7 </template>
8 <script>
9   import School from './components/School.vue'
10  import Student from './components/Student.vue'
11  export default {
12    name: 'App',
13    components: {
14      School,
15      Student
16    }
17  }
18 </script>
19 <style>
20 #app {
21   font-family: Avenir, Helvetica, Arial, sans-serif;
22   -webkit-font-smoothing: antialiased;
23   -moz-osx-font-smoothing: grayscale;
```

```
24   text-align: center;
25   color: #2c3e50;
26   margin-top: 60px;
27 }
28 </style>
29
```

main.js:

入口文件

```
1  import Vue from 'vue'
2  import App from './App.vue'
3  Vue.config.productionTip = false
4  new Vue({
5    render: h => h(App),
6  }).$mount('#app')
7
```

接下来就要详细讲解 main.js 中的 render 函数

render函数

插入一个小知识：使用 import 导入第三方库的时候不需要加 './' 导入我们自己写的：

```
1  import App from './App.vue'
2
```

导入第三方的

```
1  import Vue from 'vue'
2
```

不需要在 from 'vue' 加 './' 的原因是第三方库 node_modules 人家帮我们配置好了。
我们通过 import 导入第三方库，在第三方库的 package.json 文件中确定了我们引入的是哪个文件

通过 module 确定了我们引入的文件。


```

131  ✓    "*.js": [
132      |      "eslint --fix",
133      |      "git add"
134      |    ],
135  },
136  "main": "dist/vue.runtime.common.js",
137  "module": "dist/vue.runtime.esm.js",
138  "name": "vue",
139  ✓  "repository": {
140      |    "type": "git",
141      |    "url": "git+https://github.com/vuejs/vue.git"
142      |  },
    > 调试

```

CSDN @格雷狐思

回到 render 函数 之前的写法是这样：

```

1  import App from './App.vue'
2  new Vue({
3    el: '#root',
4    template: `<App></App>`,
5    components: {App},
6  })
7

```

如果这样子写，运行的话会引发如下的报错

报错的意思是，是在使用运行版本的 vue，没有模板解析器。

从上面的小知识可以知道，我们引入的 vue 不是完整版的，是残缺的（为了减小vue的大小）。所以残缺的vue.js 只有通过 render 函数才能把项目给跑起来。

来解析一下render

```

[HMR] Waiting for update signal from WDS...
[Vue warn]: Cannot find element: #root
[Vue warn]: You are using the runtime-only build of Vue where the template compiler is not available. Either pre-compile the templates into render functions, or use the compiler-included build.
(found in <Root>)
You are running Vue in development mode.
Make sure to turn on production mode when deploying for production.
See more tips at https://vuejs.org/guide/deployment.html

```

CSDN @格雷狐思

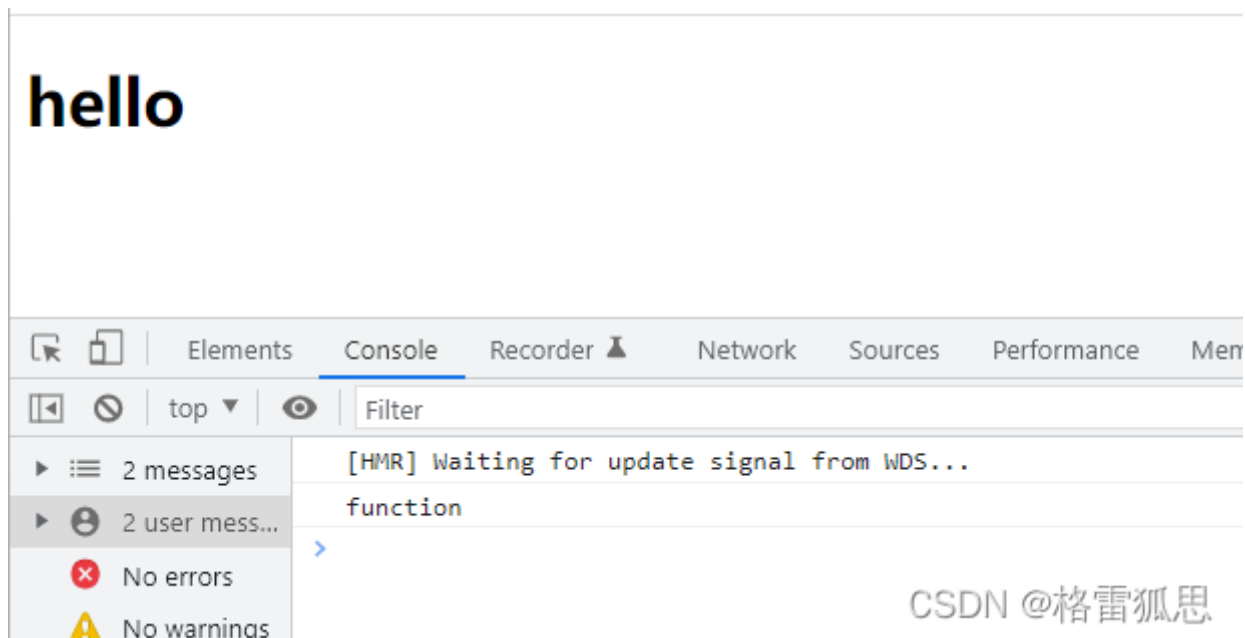
1 // render最原始写的方式

```

2 // render是个函数，还能接收到参数a
3 // 这个 createElement 很关键，是个回调函数
4 new Vue({
5   render(createElement) {
6     console.log(typeof createElement);
7     // 这个 createElement 回调函数能创建元素
8     // 因为残缺的vue 不能解析 template，所以render就来帮忙解决这个问题
9     // createElement 能创建具体的元素
10    return createElement('h1', 'hello')
11  }
12 }).$mount('#app')
13

```

因为 render 函数内并没有用到 this，所以可以简写成箭头函数。



```

1 new Vue({
2   // render: h => h(App),
3   render: (createElement) => {
4     return createElement(App)
5   }
6 }).$mount('#app')
7

```

再简写：

```

1 new Vue({
2   // render: h => h(App),

```

```
3   render: createElement => createElement(App)
4   }).$mount('#app')
5
```

最后把 createElement 换成 h 就完事了。

算啦算啦，把简写都整理一遍吧，js里的简写确实多哇。

对象内写方法最原始的：

```
1 let obj = {
2   name: 'aaa',
3   work: function (salary) {
4       return '工资' + salary;
5   }
6 }
7
```

ES6 简化版：

```
1 let obj = {
2   name: 'aaa',
3   work(salary) {
4       return '工资' + salary;
5   }
6 }
7
```

箭头函数简化版：

```
1 let obj = {
2   name: 'aaa',
3   work: (salary) => {
4       return '工资' + salary;
5   }
6 }
7
```

箭头函数再简化（最终版）：

```
1 // 只有一个参数就可以把圆括号去掉了，函数体内部只有一个 return 就可以把大括号去掉，return去掉
```

```
2 let obj = {
3   name: 'aaa',
4   work: salary => '工资' + salary;
5 }
6
```

这样就可以理解 render 函数的简写方式了。

来个不同版本 vue 的区别

- vue.js与vue.runtime.xxx.js的区别：
 - vue.js是完整版的Vue，包含：核心功能+模板解析器。
 - vue.runtime.xxx.js是运行版的Vue，只包含：核心功能；没有模板解析器。
- 因为vue.runtime.xxx.js没有模板解析器，所以不能使用template配置项，需要使用render函数接收到的createElement函数去指定具体内容。

修改脚手架的默认配置

- 使用vue inspect > output.js可以查看到Vue脚手架的默认配置。
- 使用vue.config.js可以对脚手架进行个性化定制，详情见：<https://cli.vuejs.org/zh>

脚手架中的index.html

```
1 <!DOCTYPE html>
2 <html lang="">
3   <head>
4     <meta charset="utf-8">
5     <!-- 针对IE浏览器的一个特殊配置，含义是让IE浏览器以最高的渲染级别渲染页面 -->
6     <meta http-equiv="X-UA-Compatible" content="IE=edge">
7     <!-- 开启移动端的理想视口 -->
8     <meta name="viewport" content="width=device-width,initial-scale=1.0">
9     <!-- 配置页签图标 -->
10    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
11    <!-- 引入第三方样式 -->
12    <link rel="stylesheet" href="<%= BASE_URL %>css/bootstrap.css">
13    <!-- 配置网页标题 -->
14    <title>硅谷系统</title>
15  </head>
16  <body>
17
18    <!-- 当浏览器不支持js时noscript中的元素就会被渲染 -->
19    <noscript>
```

```

19     <strong>We're sorry but <%= htmlWebpackPlugin.options.title %> doesn't work
    properly without JavaScript enabled. Please enable it to continue.</strong>
20   </noscript>
21     <!-- 容器 -->
22     <div id="app"></div>
23     <!-- built files will be auto injected -->
24   </body>
25 </html>
26

```

2.2 vue 零碎的一些知识

ref属性

- 被用来给元素或子组件注册引用信息（id的替代者）
- 应用在html标签上获取的是真实DOM元素，应用在组件标签上是组件实例对象（vc）
- 使用方式：
 - 打标识：

<h1 ref="xxx">.....</h1>或

<School ref="xxx"> </School>

- 获取：

this.\$refs.xxx

具体案例

```

1  <template>
2    <div>
3
4      <h1 v-text="msg" ref="title"></h1>
5      <button ref="btn" @click="showDOM">点我输出上方的DOM元素</button>
6      <School ref="sch"/>
7    </div>
8  </template>
9  <script>
10    //引入School组件
11    import School from './components/School'
12    export default {
13      name: 'App',
14      components:{School},
15      data() {
16        return {

```

```

16             msg: '欢迎学习Vue! '
17         }
18     },
19     methods: {
20         showDOM(){
21             console.log(this.$refs.title) //真实DOM元素
22             console.log(this.$refs.btn) //真实DOM元素
23             console.log(this.$refs.sch) //School组件的实例对象（vc）
24         }
25     },
26 }
27 </script>
28

```

props配置项

1. 功能：让组件接收外部传过来的数据

2. 传递数据：

<Demo name="xxx"/>

3. 接收数据：

a. 第一种方式（只接收）：

props:['name']

b. 第二种方式（限制类型）：

props:{name:String}

c. 第三种方式（限制类型、限制必要性、指定默认值）：

```

1  props:{
2    name:{
3      type:String, //类型
4      required:true, //必要性
5      default:'老王' //默认值
6    }
7  }
8

```

备注：props是只读的，Vue底层会监测你对props的修改，如果进行了修改，就会发出警告，若业务需求确实需要修改，那么请复制props的内容到data中一份，然后去修改data中的数据。示

例代码：父组件给子组件传数据 App.vue

```
1 <template>
2   <div id="app">
3     
4     <Student></Student>
5     <School name="haha" :age="this.age"></School>
6   </div>
7 </template>
8 <script>
9 import School from './components/School.vue'
10 import Student from './components/Student.vue'
11 export default {
12   name: 'App',
13   data () {
14     return {
15       age: 360
16     }
17   },
18   components: {
19     School,
20     Student
21   }
22 }
23 </script>
24 <style>
25 #app {
26   font-family: Avenir, Helvetica, Arial, sans-serif;
27   -webkit-font-smoothing: antialiased;
28   -moz-osx-font-smoothing: grayscale;
29   text-align: center;
30   color: #2c3e50;
31   margin-top: 60px;
32 }
33 </style>
34
```

School.vue

```
1 <template>
2   <div class="demo">
3     <h2>学校名称: {{ name }}</h2>
4     <h2>学校年龄: {{ age }}</h2>
5     <h2>学校地址: {{ address }}</h2>
6     <button @click="showName">点我提示学校名</button>
7   </div>
8 </template>
9 <script>
10 export default {
11   name: "School",
12   // 最简单的写法: props: ['name', 'age']
13   props: {
14     name: {
15       type: String,
16       required: true // 必须要传的
17     },
18     age: {
19       type: Number,
20       required: true
21     }
22   },
23   data() {
24     return {
25       address: "北京昌平",
26     };
27   },
28   methods: {
29     showName() {
30       alert(this.name);
31     },
32   },
33 };
34 </script>
35 <style>
36 .demo {
37   background-color: orange;
38 }
39 </style>
```


mixin(混入)

混入 (mixin) 提供了一种非常灵活的方式，来分发 Vue 组件中的可复用功能。一个混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被“混合”进入该组件本身的选项。

例子：

```
1 // 定义一个混入对象
2 var myMixin = {
3   created: function () {
4     this.hello()
5   },
6   methods: {
7     hello: function () {
8       console.log('hello from mixin!')
9     }
10  }
11 }
12 // 定义一个使用混入对象的组件
13 var Component = Vue.extend({
14   mixins: [myMixin]
15 })
16
```

选项合并

当组件和混入对象含有同名选项时，这些选项将以恰当的方式进行“合并”。

比如，数据对象在内部会进行递归合并，并在发生冲突时以组件数据优先。

```
1 var mixin = {
2   data: function () {
3     return {
4       message: 'hello',
5       foo: 'abc'
6     }
7   }
8 }
9 new Vue({
10   mixins: [mixin],

```

```

11 data: function () {
12   return {
13     message: 'goodbye',
14     bar: 'def'
15   }
16 },
17 created: function () {
18   console.log(this.$data)
19   // => { message: "goodbye", foo: "abc", bar: "def" }
20 }
21 })
22

```

同名钩子函数将合并为一个数组，因此都将被调用。另外，混入对象的钩子将在组件自身钩子**之前**调用。

```

1 var mixin = {
2   created: function () {
3     console.log('混入对象的钩子被调用')
4   }
5 }
6 new Vue({
7   mixins: [mixin],
8   created: function () {
9     console.log('组件钩子被调用')
10   }
11 })
12 // => "混入对象的钩子被调用"
13 // => "组件钩子被调用"
14

```

值为对象的选项，例如 `methods`、`components` 和 `directives`，将被合并为同一个对象。两个对象键名冲突时，取组件对象的键值对。

```

1 var mixin = {
2   methods: {
3     foo: function () {
4       console.log('foo')
5     },

```

```

6     conflicting: function () {
7         console.log('from mixin')
8     }
9 }
10 }
11 var vm = new Vue({
12     mixins: [mixin],
13     methods: {
14         bar: function () {
15             console.log('bar')
16         },
17         conflicting: function () {
18             console.log('from self')
19         }
20     }
21 })
22 vm.foo() // => "foo"
23 vm.bar() // => "bar"
24 vm.conflicting() // => "from self"
25

```

全局混入不建议使用

插件

插件通常用来为 Vue 添加全局功能。插件的功能范围没有严格的限制。

通过全局方法 `Vue.use()` 使用插件。它需要在你调用 `new Vue()` 启动应用之前完成：

```

1 // 调用 `MyPlugin.install(Vue)`
2 Vue.use(MyPlugin)
3 new Vue({
4     // ...组件选项
5 })
6

```

本质：包含 `install` 方法的一个对象，`install` 的第一个参数是 `Vue`，第二个以后的参数是插件使用者传递的数据。

定义插件：

```

1 对象.install = function (Vue, options) {

```

```

2    // 1. 添加全局过滤器
3    Vue.filter(....)
4    // 2. 添加全局指令
5    Vue.directive(....)
6    // 3. 配置全局混入(合)
7    Vue.mixin(....)
8    // 4. 添加实例方法
9    Vue.prototype.$myMethod = function () {...}
10   Vue.prototype.$myProperty = xxxx
11 }
12

```

具体案例:

plugin.js

```

1  export default {
2    install(Vue, x, y, z) {
3      console.log(x, y, z)
4      //全局过滤器
5      Vue.filter('mySlice', function (value) {
6        return value.slice(0, 4)
7      })
8      //定义全局指令
9      Vue.directive('fbind', {
10        //指令与元素成功绑定时（一上来）
11        bind(element, binding) {
12          element.value = binding.value
13        },
14        //指令所在元素被插入页面时
15        inserted(element, binding) {
16          element.focus()
17        },
18        //指令所在的模板被重新解析时
19        update(element, binding) {
20          element.value = binding.value
21        }
22      })
23      //定义混入
24      Vue.mixin({

```

```

25         data() {
26             return {
27                 x: 100,
28                 y: 200
29             }
30         },
31     })
32     //给Vue原型上添加一个方法（vm和vc就都能用了）
33     Vue.prototype.hello = () => { alert('你好啊aaaa') }
34 }
35 }
36

```

main.js

```

1  // 引入插件
2  import plugin from './plugin'
3  // 使用插件
4  Vue.use(plugin)
5

```

然后就可以在别的组件使用插件里的功能了。

scoped样式

1. 作用：让样式在局部生效，防止冲突。
2. 写法：

<style scoped>

具体案例：

```

1  <style lang="less" scoped>
2      .demo{
3          background-color: pink;
4          .atguigu{
5              font-size: 40px;
6          }
7      }
8  </style>
9

```

总结TodoList案例

1. 组件化编码流程：

- (1).拆分静态组件：组件要按照功能点拆分，命名不要与html元素冲突。
- (2).实现动态组件：考虑好数据的存放位置，数据是一个组件在用，还是一些组件在用：
 - 1).一个组件在用：放在组件自身即可。
 - 2). 一些组件在用：放在他们共同的父组件上（状态提升）。
- (3).实现交互：从绑定事件开始。

2. props适用于：

- (1).父组件 ==> 子组件 通信
 - (2).子组件 ==> 父组件 通信（要求父先给子一个函数）
3. 使用v-model时要切记：v-model绑定的值不能是props传过来的值，因为props是不可以修改的！
4. props传过来的若是对象类型的值，修改对象中的属性时Vue不会报错，但不推荐这样做。

2.3 浏览器本地存储

Cookie

Cookie是最早被提出来的本地存储方式，在此之前，服务端是无法判断网络中的两个请求是否是同一用户发起的，为解决这个问题，Cookie就出现了。Cookie 是存储在用户浏览器中的一段不超过 4 KB 的字符串。它由一个名称（Name）、一个值（Value）和其它几个用于控制 Cookie 有效期、安全性、使用范围的可选属性组成。不同域名下的 Cookie 各自独立，每当客户端发起请求时，会自动把当前域名下所有未过期的 Cookie 一同发送到服务器。

Cookie的特性：

- Cookie一旦创建成功，名称就无法修改
- Cookie是无法跨域名的，也就是说a域名和b域名下的cookie是无法共享的，这也是由Cookie的隐私安全性决定的，这样就能够阻止非法获取其他网站的Cookie
- 每个域名下Cookie的数量不能超过20个，每个Cookie的大小不能超过4kb
- 有安全问题，如果Cookie被拦截了，那就可获得session的所有信息，即使加密也于事无补，无需知道cookie的意义，只要转发cookie就能达到目的
- Cookie在请求一个新的页面的时候都会被发送过去

Cookie 在身份认证中的作用

客户端第一次请求服务器的时候，服务器通过响应头的形式，向客户端发送一个身份认证的 Cookie，客户端会自动将 Cookie 保存在浏览器中。

随后，当客户端浏览器每次请求服务器的时候，浏览器会自动将身份认证相关的 Cookie，通过请求头的形式发送给服务器，服务器即可验明客户端的身份。

Cookie 不具有安全性

由于 Cookie 是存储在浏览器中的，而且浏览器也提供了读写 Cookie 的 API，因此 Cookie 很容易被伪造，不具有安全性。因此不建议服务器将重要的隐私数据，通过 Cookie 的形式发送给浏览器。

注意：千万不要使用 Cookie 存储重要且隐私的数据！比如用户的身份信息、密码等。

Session

Session是另一种记录客户状态的机制，不同的是Cookie保存在客户端浏览器中，而Session保存在服务器上。客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上。这就是 Session。客户端浏览器再次访问时只需要从该Session中查找该客户的状态就可以了session是一种特殊的cookie。cookie是保存在客户端的，而session是保存在服务端。

为什么要用session

由于cookie 是存在用户端，而且它本身存储的尺寸大小也有限，最关键是可以是可见的，并可以随意的修改，很不安全。那如何又要安全，又可以方便的全局读取信息呢？于是，这个时候，一种新的存储会话机制：session 诞生了

session原理

当客户端第一次请求服务器的时候，服务器生成一份session保存在服务端，将该数据(session)的id以cookie的形式传递给客户端；以后的每次请求，浏览器都会自动的携带cookie来访问服务器(session数据id)。

图示：session我觉得可以简单理解为一个表，根据cookie传来的值查询表中的内容 **session 标准工作流程** 我在 node.js 中详细演示了一遍 session 的使用，具体看了另一篇博客：<https://blog.csdn.net/hangao233/article/details/123089029>

LocalStorage

LocalStorage是HTML5新引入的特性，由于有的时候我们存储的信息较大，Cookie就不能满足我们的需求，这时候LocalStorage就派上用场了。

LocalStorage的优点：

- 在大小方面，LocalStorage的大小一般为5MB，可以储存更多的信息
- LocalStorage是持久储存，并不会随着页面的关闭而消失，除非主动清理，不然会永久存在
- 仅储存在本地，不像Cookie那样每次HTTP请求都会被携带

LocalStorage的缺点：

- 存在浏览器兼容问题，IE8以下版本的浏览器不支持
- 如果浏览器设置为隐私模式，那我们将无法读取到LocalStorage

- LocalStorage受到同源策略的限制，即端口、协议、主机地址有任何一个不相同，都不会访问

LocalStorage的常用API:

```
1 // 保存数据到 localStorage
2 localStorage.setItem('key', 'value');
3 // 从 localStorage 获取数据
4 let data = localStorage.getItem('key');
5 // 从 localStorage 删除保存的数据
6 localStorage.removeItem('key');
7 // 从 localStorage 删除所有保存的数据
8 localStorage.clear();
9 // 获取某个索引的Key
10 localStorage.key(index)
11
```

LocalStorage的使用场景:

- 有些网站有换肤的功能，这时候就可以将换肤的信息存储在本地的LocalStorage中，当需要换肤的时候，直接操作LocalStorage即可
- 在网站中的用户浏览信息也会存储在LocalStorage中，还有网站的一些不常变动的个人信息等也可以存储在本地的LocalStorage中

SessionStorage

SessionStorage和LocalStorage都是在HTML5才提出来的存储方案，SessionStorage 主要用于临时保存同一窗口(或标签页)的数据，刷新页面时不会删除，关闭窗口或标签页之后将会删除这些数据。

SessionStorage与LocalStorage对比:

- SessionStorage和LocalStorage都在**本地进行数据存储**;
- SessionStorage也有同源策略的限制，但是SessionStorage有一条更加严格的限制，SessionStorage**只有在同一浏览器的同一窗口下才能够共享**;
- LocalStorage和SessionStorage**都不能被爬虫爬取**;

SessionStorage的常用API:

```
1 // 保存数据到 sessionStorage
2 sessionStorage.setItem('key', 'value');
3 // 从 sessionStorage 获取数据
4 let data = sessionStorage.getItem('key');
5 // 从 sessionStorage 删除保存的数据
6 sessionStorage.removeItem('key');
```



```
7 // 从 sessionStorage 删除所有保存的数据
8 sessionStorage.clear();
9 // 获取某个索引的Key
10 sessionStorage.key(index)
11
```

SessionStorage的使用场景

由于SessionStorage具有时效性，所以可以用来存储一些网站的游客登录的信息，还有临时的浏览记录的信息。当关闭网站之后，这些信息也就随之消除了。

具体案例：localStorage

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8" />
5     <title>localStorage</title>
6   </head>
7   <body>
8     <h2>localStorage</h2>
9     <button onclick="saveData()">点我保存一个数据</button>
10    <button onclick="readData()">点我读取一个数据</button>
11    <button onclick="deleteData()">点我删除一个数据</button>
12    <button onclick="deleteAllData()">点我清空一个数据</button>
13    <script type="text/javascript" >
14      let p = {name:'张三',age:18}
15      function saveData(){
16        localStorage.setItem('msg','hello!!!')
17        localStorage.setItem('msg2',666)
18        // 转成 JSON 对象存进去
19        localStorage.setItem('person',JSON.stringify(p))
20      }
21      function readData(){
22        console.log(localStorage.getItem('msg'))
23        console.log(localStorage.getItem('msg2'))
24        const result = localStorage.getItem('person')
25        console.log(JSON.parse(result))
26        // console.log(localStorage.getItem('msg3'))
27      }
28      function deleteData(){
```

```

29         localStorage.removeItem('msg2')
30     }
31     function deleteAllData(){
32         localStorage.clear()
33     }
34 </script>
35 </body>
36 </html>
37

```

sessionStorage

```

1  <!DOCTYPE html>
2  <html>
3    <head>
4
5      <meta charset="UTF-8" />
6      <title>sessionStorage</title>
7    </head>
8    <body>
9
10     <h2>sessionStorage</h2>
11     <button onclick="saveData()">点我保存一个数据</button>
12     <button onclick="readData()">点我读取一个数据</button>
13     <button onclick="deleteData()">点我删除一个数据</button>
14     <button onclick="deleteAllData()">点我清空一个数据</button>
15     <script type="text/javascript" >
16       let p = {name:'张三',age:18}
17       function saveData(){
18         sessionStorage.setItem('msg','hello!!!')
19         sessionStorage.setItem('msg2',666)
20         // 转换成JSON 字符串存进去
21         sessionStorage.setItem('person',JSON.stringify(p))
22       }
23       function readData(){
24         console.log(sessionStorage.getItem('msg'))
25         console.log(sessionStorage.getItem('msg2'))
26         const result = sessionStorage.getItem('person')
27         console.log(JSON.parse(result))
28         // console.log(sessionStorage.getItem('msg3'))
29       }
30     </script>
31   </body>
32 </html>

```

```

28         function deleteData(){
29             sessionStorage.removeItem('msg2')
30         }
31         function deleteAllData(){
32             sessionStorage.clear()
33         }
34     </script>
35 </body>
36 </html>
37

```

2.4 组件自定义事件

组件自定义事件是一种组件间通信的方式，适用于：<strong style="color:red">子组件 ==> 父组件

使用场景

A是父组件，B是子组件，B想给A传数据，那么就要在A中给B绑定自定义事件（事件的回调在A中）。

绑定自定义事件：

第一种方式，在父组件中：<Demo @atguigu="test"/>或 <Demo v-on:atguigu="test"/>

具体代码 App.vue

```

1  <template>
2    <div class="app">
3      <!-- 通过父组件给子组件绑定一个自定义事件实现：子给父传递数据（第一种写法，使用@或v-on） -->
4      <Student @atguigu="getStudentName"/>
5    </div>
6  </template>
7  <script>
8    import Student from './components/Student'
9    export default {
10      name: 'App',
11      components: {Student},
12      data() {
13        return {
14          msg: '你好啊！',
15          studentName: ''
16        }

```

```

17         },
18         methods: {
19             getStudentName(name,...params){
20                 console.log('App收到了学生名: ',name,params)
21                 this.studentName = name
22             }
23         }
24     }
25 </script>
26 <style scoped>
27     .app{
28         background-color: gray;
29         padding: 5px;
30     }
31 </style>
32

```

Student.vue

```

1 <template>
2 <div class="student">
3     <button @click="sendStudentName">把学生名给App</button>
4 </div>
5 </template>
6 <script>
7     export default {
8         name: 'Student',
9         data() {
10             return {
11                 name: '张三',
12             }
13         },
14         methods: {
15             sendStudentName(){
16                 //触发Student组件实例身上的atguigu事件
17                 this.$emit('atguigu',this.name,666,888,900)
18             }
19         },
20     }

```

```

21 </script>
22 <style lang="less" scoped>
23     .student{
24         background-color: pink;
25         padding: 5px;
26         margin-top: 30px;
27     }
28 </style>
29

```

第二种方式，在父组件中：

使用 `this.$refs.xxx.$on()` 这样写起来更灵活，比如可以加定时器啥的。

具体代码 App.vue

```

1 <template>
2   <div class="app">
3       <!-- 通过父组件给子组件绑定一个自定义事件实现：子给父传递数据（第二种写法，使用ref） -->
4       <Student ref="student"/>
5   </div>
6 </template>
7 <script>
8     import Student from './components/Student'
9     export default {
10         name: 'App',
11         components: {Student},
12         data() {
13             return {
14                 studentName: ''
15             }
16         },
17         methods: {
18             getStudentName(name, ...params){
19                 console.log('App收到了学生名: ', name, params)
20                 this.studentName = name
21             },
22         },
23         mounted() {
24             this.$refs.student.$on('atguigu', this.getStudentName) //绑定自定义事件

```

```

25 // this.$refs.student.$once('atguigu',this.getStudentName) //绑定自定义事件（一次性）
26 },
27 }
28 </script>
29 <style scoped>
30 .app{
31     background-color: gray;
32     padding: 5px;
33 }
34 </style>
35

```

Student.vue

```

1 <template>
2 <div class="student">
3     <button @click="sendStudentName">把学生名给App</button>
4 </div>
5 </template>
6 <script>
7     export default {
8         name: 'Student',
9         data() {
10             return {
11                 name: '张三',
12             }
13         },
14         methods: {
15             sendStudentName(){
16                 //触发Student组件实例身上的atguigu事件
17                 this.$emit('atguigu',this.name,666,888,900)
18             }
19         },
20     }
21 </script>
22 <style lang="less" scoped>
23     .student{
24         background-color: pink;
25         padding: 5px;

```

```

26         margin-top: 30px;
27     }
28 </style>
29

```

若想让自定义事件只能触发一次，可以使用once修饰符，或\$once方法。触发自定义事件：this.\$emit('atguigu',数据) 使用 this.\$emit() 就可以子组件向父组件传数据 **解绑自定义事件** this.\$off('atguigu') 代码

```

1  this.$off('atguigu') //解绑一个自定义事件
2  // this.$off(['atguigu','demo']) //解绑多个自定义事件
3  // this.$off() //解绑所有的自定义事件
4

```

组件上也可以绑定原生DOM事件，需要使用native修饰符。

代码

```

1  <!-- 通过父组件给子组件绑定一个自定义事件实现：子给父传递数据（第二种写法，使用ref） -->
2  <Student ref="student" @click.native="show"/>
3

```

注意：通过this.\$refs.xxx.\$on('atguigu',回调)绑定自定义事件时，回调要么配置在methods中， 要么用箭头函数，否则this指向会出问题！

2.5 全局事件总线

1. 一种组件间通信的方式，适用于任意组件间通信。
2. 安装全局事件总线：

```

1  new Vue({
2      .....
3      beforeCreate() {
4          Vue.prototype.$bus = this //安装全局事件总线，$bus就是当前应用的vm
5      },
6      .....
7  })

```

3. 使用事件总线:

- a. 接收数据: A组件想接收数据, 则在A组件中给\$bus绑定自定义事件, 事件的`回调留在A组件自身。`

```

1  methods(){
2    demo(data){.....}
3  }
4  .....
5  mounted() {
6    this.$bus.$on('xxx',this.demo)
7  }
8

```

- b. 提供数据:

`this.$bus.$emit('xxx',数据)`

4. 最好在beforeDestroy钩子中, 用\$off去解绑`当前组件所用到的`事件。

示例代码 School.vue

```

1  <template>
2    <div class="school">
3      <h2>学校名称: {{name}}</h2>
4      <h2>学校地址: {{address}}</h2>
5    </div>
6  </template>
7  <script>
8    export default {
9      name: 'School',
10     data() {
11       return {
12         name: '尚硅谷',
13         address: '北京',
14       }
15     },
16     methods: {

```



```

17         demo(data) {
18             console.log('我是School组件，收到了数据',data)
19         }
20     }
21     mounted() {
22         // console.log('School',this)
23         this.$bus.$on('hello',this.demo)
24     },
25     beforeDestroy() {
26         this.$bus.$off('hello')
27     },
28 }
29 </script>
30 <style scoped>
31     .school{
32         background-color: skyblue;
33         padding: 5px;
34     }
35 </style>
36

```

Student.vue

```

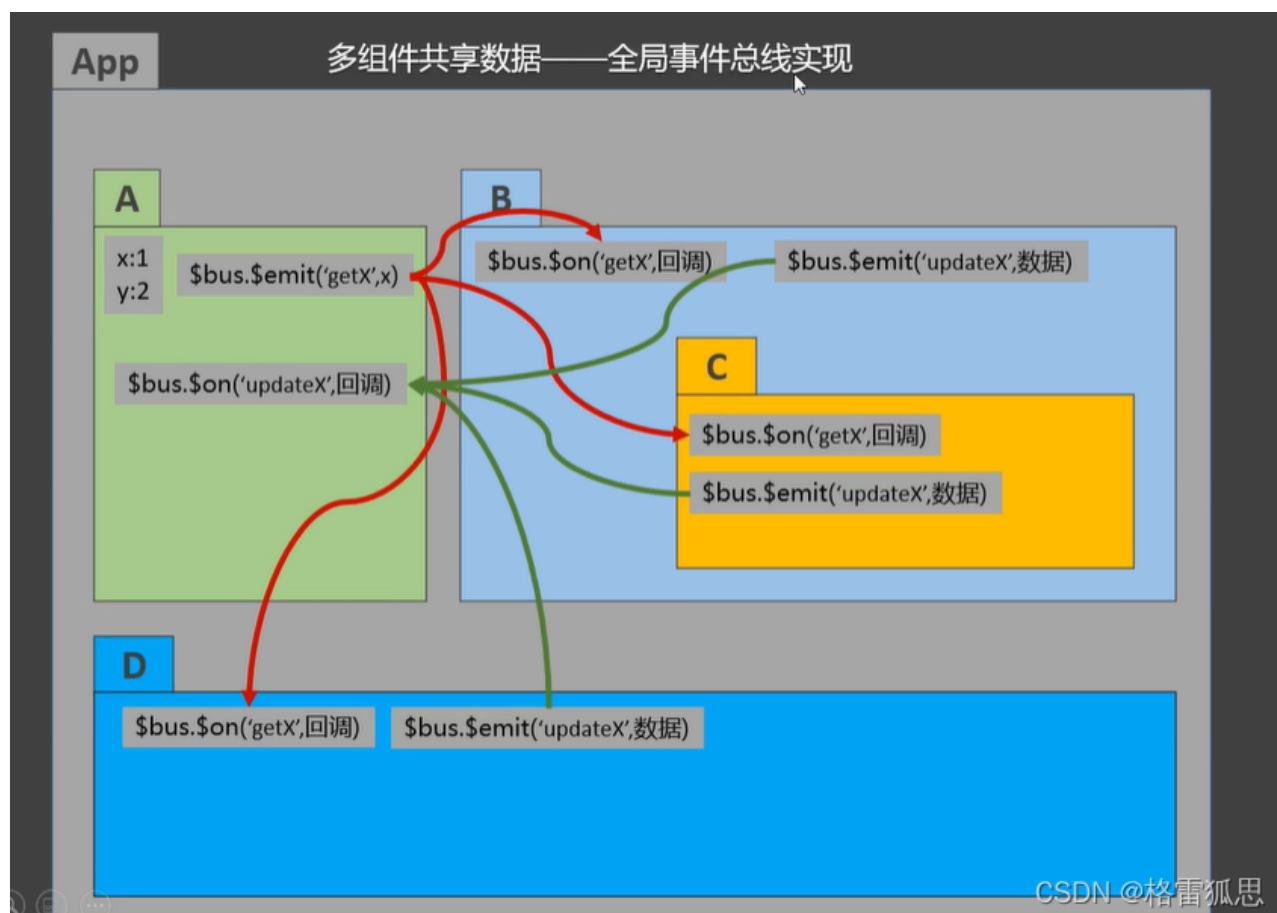
1  <template>
2    <div class="student">
3        <h2>学生姓名: {{name}}</h2>
4        <h2>学生性别: {{sex}}</h2>
5        <button @click="sendStudentName">把学生名给School组件</button>
6    </div>
7  </template>
8  <script>
9      export default {
10         name: 'Student',
11         data() {
12             return {
13                 name: '张三',
14                 sex: '男',
15             }
16         },

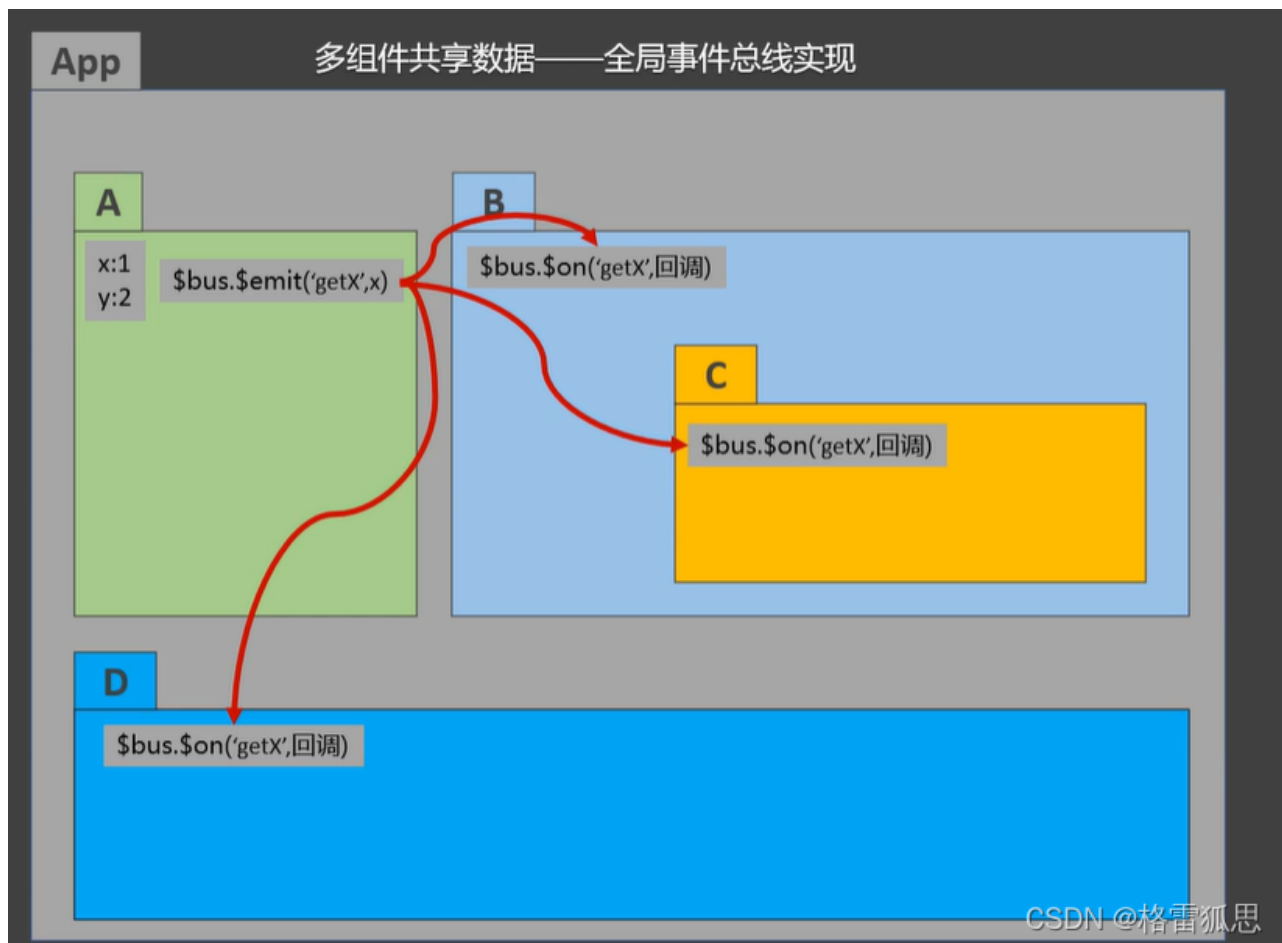
```

```

17         mounted() {
18             // console.log('Student',this.x)
19         },
20         methods: {
21             sendStudentName(){
22                 this.$bus.$emit('hello',this.name)
23             }
24         },
25     }
26 </script>
27 <style lang="less" scoped>
28     .student{
29         background-color: pink;
30         padding: 5px;
31         margin-top: 30px;
32     }
33 </style>
34

```





2.6 消息订阅与发布

1. 一种组件间通信的方式，适用于任意组件间通信。

2. 使用步骤：

a. 安装pubsub：

`npm i pubsub-js`

b. 引入：

`import pubsub from 'pubsub-js'`

c. 接收数据：A组件想接收数据，则在A组件中订阅消息，订阅的回调留在A组件自身。

```
1  methods:{
2    demo(data){.....}
3  }
4  .....
5  mounted() {
6    this.pid = pubsub.subscribe('xxx', this.demo) //订阅消息
```

```
7 }
```

```
8
```

d. 提供数据：

pubsub.publish('xxx',数据)

e. 最好在beforeDestroy钩子中，用

PubSub.unsubscribe(pid)去取消订阅。

示例代码 订阅消息 **School.vue**

```
1 <template>
2   <div class="school">
3       <h2>学校名称: {{name}}</h2>
4       <h2>学校地址: {{address}}</h2>
5   </div>
6 </template>
7 <script>
8     import pubsub from 'pubsub-js'
9     export default {
10         name: 'School',
11         data() {
12             return {
13                 name: '尚硅谷',
14                 address: '北京',
15             }
16         },
17         mounted() {
18             // console.log('School',this)
19             /* this.$bus.$on('hello',(data)=>{
20                 console.log('我是School组件，收到了数据',data)
21             }) */
22             this.pubId = pubsub.subscribe('hello',(msgName,data)=>{
23                 console.log(this)
24                 // console.log('有人发布了hello消息，hello消息的回调执行
了',msgName,data)
25             })
26         },
27         beforeDestroy() {
28             // this.$bus.$off('hello')
29             pubsub.unsubscribe(this.pubId)
```

```

30         },
31     }
32 </script>
33 <style scoped>
34     .school{
35         background-color: skyblue;
36         padding: 5px;
37     }
38 </style>
39

```

发布消息

Student.vue

```

1  <template>
2    <div class="student">
3        <h2>学生姓名: {{name}}</h2>
4        <h2>学生性别: {{sex}}</h2>
5        <button @click="sendStudentName">把学生名给School组件</button>
6    </div>
7  </template>
8  <script>
9      import pubsub from 'pubsub-js'
10     export default {
11         name: 'Student',
12         data() {
13             return {
14                 name: '张三',
15                 sex: '男',
16             }
17         },
18         mounted() {
19             // console.log('Student',this.x)
20         },
21         methods: {
22             sendStudentName(){
23                 // this.$bus.$emit('hello',this.name)
24                 pubsub.publish('hello',666)
25             }
26         }
27     }
28 }

```

```
26         },
27     }
28 </script>
29 <style lang="less" scoped>
30     .student{
31         background-color: pink;
32         padding: 5px;
33         margin-top: 30px;
34     }
35 </style>
36
```

2.7 nextTick

1. 语法：

`this.$nextTick(回调函数)`

2. 作用：在下次 DOM 更新结束后执行其指定的回调。

3. 什么时候用：当改变数据后，要基于更新后的新DOM进行某些操作时，要在nextTick所指定的回调函数中执行。

具体案例

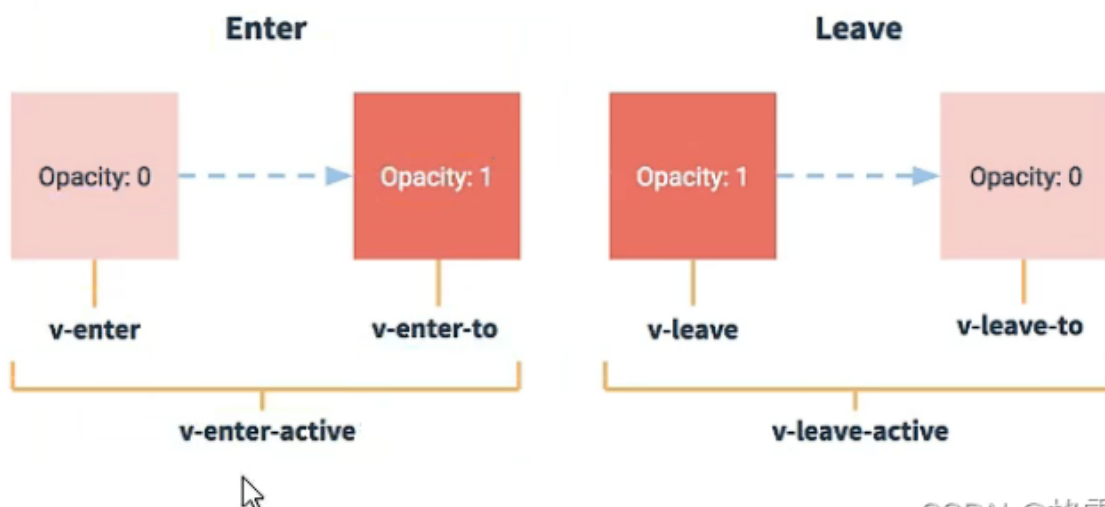
```
1  this.$nextTick(function() {
2    this.$refs.inputTitle.focus()
3  })
4
```

2.8 Vue封装的过度与动画

1. 作用：在插入、更新或移除 DOM元素时，在合适的时候给元素添加样式类名。

2. 图示：

4. 写法：



CSDN @格雷狐思

1. 准备好样式：

- 元素进入的样式：
 - i. v-enter: 进入的起点
 - ii. v-enter-active: 进入过程中
 - iii. v-enter-to: 进入的终点
- 元素离开的样式：
 - i. v-leave: 离开的起点
 - ii. v-leave-active: 离开过程中
 - iii. v-leave-to: 离开的终点

2. 使用

<transition>包裹要过渡的元素，并配置name属性：

```
1 <transition name="hello">
2   <h1 v-show="isShow">你好啊！ </h1>
3 </transition>
4
```

3. 备注：若有多个元素需要过度，则需要使用：

<transition-group>，且每个元素都要指定key值。

具体案例（单个元素过渡）

```
1 <template>
2   <div>
3     <button @click="isShow = !isShow">显示/隐藏</button>
```

```

4         <transition appear>
5             <h1 v-show="isShow">你好啊! </h1>
6         </transition>
7     </div>
8 </template>
9 <script>
10     export default {
11         name: 'Test',
12         data() {
13             return {
14                 isShow: true
15             }
16         },
17     }
18 </script>
19 <style scoped>
20     h1{
21         background-color: orange;
22     }
23     .v-enter-active{
24         animation: move 0.5s linear;
25     }
26     .v-leave-active{
27         animation: move 0.5s linear reverse;
28     }
29     @keyframes move {
30         from{
31             transform: translateX(-100%);
32         }
33         to{
34             transform: translateX(0px);
35         }
36     }
37 </style>
38

```

name 的作用可以让不同的元素有不同的动画效果

1 <template>


```

2  <div>
3      <button @click="isShow = !isShow">显示/隐藏</button>
4      <transition name="hello" appear>
5          <h1 v-show="isShow">你好啊! </h1>
6      </transition>
7  </div>
8  </template>
9  <script>
10      export default {
11          name: 'Test',
12          data() {
13              return {
14                  isShow: true
15              }
16          },
17      }
18  </script>
19  <style scoped>
20      h1{
21          background-color: orange;
22      }
23      .hello-enter-active{
24          animation: move 0.5s linear;
25      }
26      .hello-leave-active{
27          animation: move 0.5s linear reverse;
28      }
29      @keyframes move {
30          from{
31              transform: translateX(-100%);
32          }
33          to{
34              transform: translateX(0px);
35          }
36      }
37  </style>
38

```

具体案例（多个元素过渡）

```

1 <template>
2   <div>
3
4       <button @click="isShow = !isShow">显示/隐藏</button>
5
6       <transition-group name="hello" appear>
7
8           <h1 v-show="!isShow" key="1">你好啊! </h1>
9
10          <h1 v-show="isShow" key="2">尚硅谷! </h1>
11
12      </transition-group>
13
14  </div>
15 </template>
16 <script>
17     export default {
18         name: 'Test',
19         data() {
20             return {
21                 isShow: true
22             }
23         },
24     }
25 </script>
26 <style scoped>
27     h1{
28
29         background-color: orange;
30
31     }
32     /* 进入的起点、离开的终点 */
33     .hello-enter,.hello-leave-to{
34
35         transform: translateX(-100%);
36
37     }
38     .hello-enter-active,.hello-leave-active{
39
40         transition: 0.5s linear;
41
42     }
43     /* 进入的终点、离开的起点 */
44     .hello-enter-to,.hello-leave{
45
46         transform: translateX(0);
47
48     }
49 </style>
50
51

```

使用第三库的具体案例（随便看看，这个不重要） 库的名称：Animate.css 安装：npm i animate.css 引入：import 'animate.css'

```

1  <template>
2    <div>
3      <button @click="isShow = !isShow">显示/隐藏</button>
4      <transition-group
5        appear
6        name="animate__animated animate__bounce"
7        enter-active-class="animate__swing"
8        leave-active-class="animate__backOutUp"
9      >
10       <h1 v-show="!isShow" key="1">你好啊! </h1>
11       <h1 v-show="isShow" key="2">尚硅谷! </h1>
12     </transition-group>
13   </div>
14 </template>
15 <script>
16   import 'animate.css'
17   export default {
18     name: 'Test',
19     data() {
20       return {
21         isShow: true
22       }
23     },
24   }
25 </script>
26 <style scoped>
27   h1{
28     background-color: orange;
29   }
30 </style>
31

```

2.9 vue脚手架配置代理

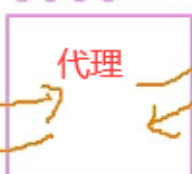
可以用来解决跨域的问题

8080



代理服务器，端口和客户端
一致就不会产生跨域

8080



代理

5000



服务器

代理服务器和真实服务器通信，服务器间的
通信不需要ajax，所以就不会产生跨域的问题

CSDN @格雷狐思

```
[HMR] waiting for update
✖ Access to XMLHttpRequest
from origin 'http://loca
'Access-Control-Allow-Or
请求失败了 Network Error
✖ ▶ GET http://localhost:5000
```

ajax 是前端技术，你得有浏览器，才有window对象，才有xhr，才能发ajax请求，服务器之间通信就用传统的http请求就行了。

方法一

在vue.config.js中添加如下配置：

```
1 devServer:{
2   proxy:"
  http://localhost:5000
  proxy:"
3 }
4
```

说明：

1. 优点：配置简单，请求资源时直接发给前端（8080）即可。
2. 缺点：不能配置多个代理，不能灵活的控制请求是否走代理。
3. 工作方式：若按照上述配置代理，当请求了前端不存在的资源时，那么该请求会转发给服务器（优先匹配前端资源）

方法二

编写vue.config.js配置具体代理规则：

```

1  module.exports = {
2    devServer: {
3      proxy: {
4        '/api1': { // 匹配所有以 '/api1'开头的请求路径
5          target: '
http://localhost:5000
          target: '
6          changeOrigin: true,
7          pathRewrite: {'^/api1': ''} //代理服务器将请求地址转给真实服务器时会将 /api1 去掉
8        },
9        '/api2': { // 匹配所有以 '/api2'开头的请求路径
10         target: '
http://localhost:5001
          target: '
11         changeOrigin: true,
12         pathRewrite: {'^/api2': ''}
13       }
14     }
15   }
16 }
17 /*
18   changeOrigin设置为true时，服务器收到的请求头中的host为：localhost:5000
19   changeOrigin设置为false时，服务器收到的请求头中的host为：localhost:8080
20   changeOrigin默认值为true
21   */
22

```

说明：

1. 优点：可以配置多个代理，且可以灵活的控制请求是否走代理。
2. 缺点：配置略微繁琐，请求资源时必须加前缀。

2.10 slot插槽

1. 作用：让父组件可以向子组件指定位置插入html结构，也是一种组件间通信的方式，适用于 `<strong style="color:red">父组件 ==> 子组件` 。
2. 分类：默认插槽、具名插槽、作用域插槽
3. 使用方式：
 - a. 默认插槽：

```

1 父组件中:
2      <Category>
3          <div>html结构1</div>
4      </Category>
5 子组件中:
6      <template>
7          <div>
8              <!-- 定义插槽 -->
9              <slot>插槽默认内容...</slot>
10         </div>
11     </template>
12

```

b. 具名插槽:

```

1 父组件中:
2      <Category>
3          <template slot="center">
4              <div>html结构1</div>
5          </template>
6
7          <template v-slot:footer>
8              <div>html结构2</div>
9          </template>
10     </Category>
11 子组件中:
12     <template>
13         <div>
14             <!-- 定义插槽 -->
15             <slot name="center">插槽默认内容...</slot>
16             <slot name="footer">插槽默认内容...</slot>
17         </div>
18     </template>
19

```

c. 作用域插槽:

- i. 理解: ``数据在组件的自身(子组件), 但根据数据生成的结构需要组件的使用者(父组件)来决定。 `` (games数据在Category(子)组件中,

但使用数据所遍历出来的结构由App（父）组件决定）

ii. 具体编码：

1 父组件中：

```
2         <Category>
3             <template scope="scopeData">
4                 <!-- 生成的是ul列表 -->
5                 <ul>
6                     <li v-for="g in scopeData.games" :key="g">{{g}}
</li>
7                 </ul>
8             </template>
9         </Category>

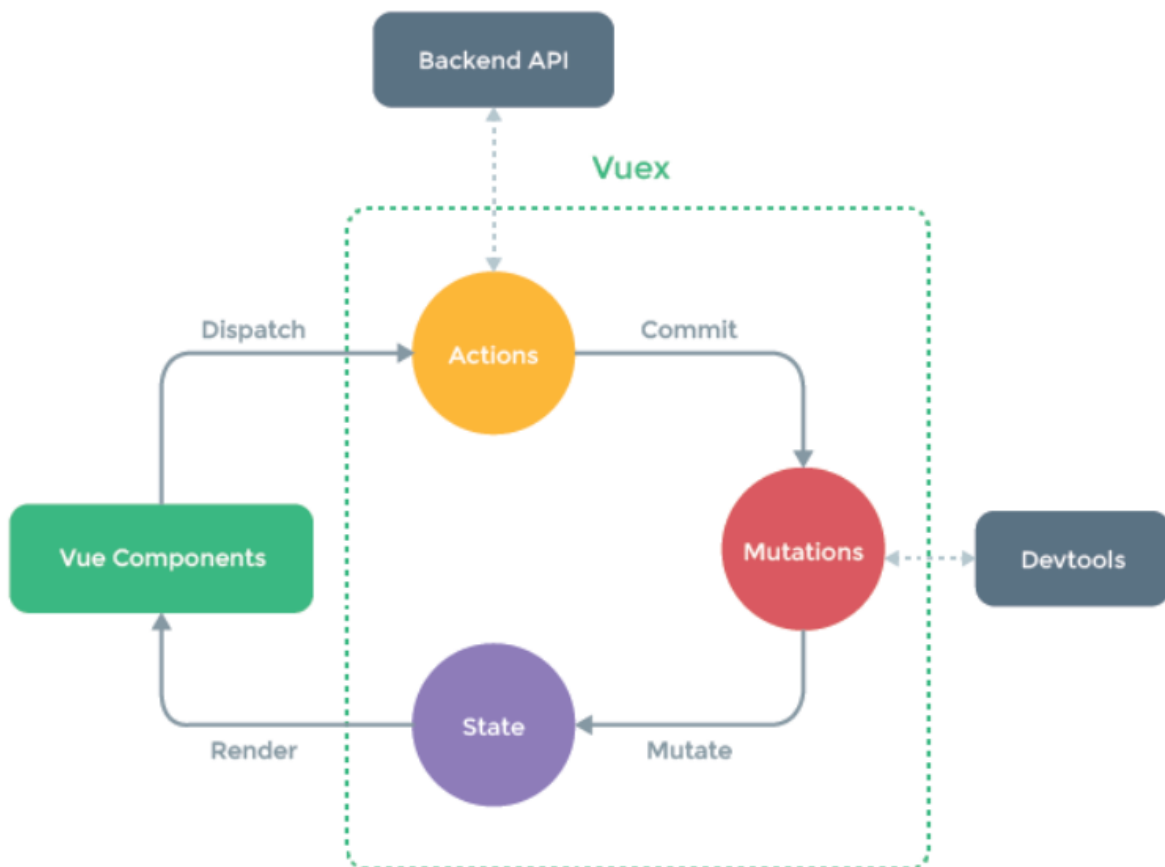
10
11         <Category>
12             <template slot-scope="scopeData">
13                 <!-- 生成的是h4标题 -->
14                 <h4 v-for="g in scopeData.games" :key="g">{{g}}</h4>
15             </template>
16         </Category>
```

17 子组件中：

```
18     <template>
19         <div>
20             <!-- 通过数据绑定就可以把子组件的数据传到父组件 -->
21             <slot :games="games"></slot>
22         </div>
23     </template>
24
25     <script>
26         export default {
27             name: 'Category',
28             props: ['title'],
29             //数据在子组件自身
30             data() {
31                 return {
32                     games: ['红色警戒', '穿越火线', '劲舞团', '超级玛丽']
33                 }
34             },
35         }
```

3. VUEX

原理图：



CSDN @格雷狐思

3.1 概念

在Vue中实现集中式状态（数据）管理的一个Vue插件，对vue应用中多个组件的共享状态进行集中式的管理（读/写），也是一种组件间通信的方式，且适用于任意组件间通信。

3.2 何时使用？

多个组件需要共享数据时

3.3 搭建vuex环境

1. 创建文件：

src/store/index.js

```
1 //引入Vue核心库
2 import Vue from 'vue'
3 //引入Vuex
4 import Vuex from 'vuex'
5 //应用Vuex插件
6 Vue.use(Vuex)
7
8 //准备actions对象—响应组件中用户的动作
9 const actions = {}
10 //准备mutations对象—修改state中的数据
11 const mutations = {}
12 //准备state对象—保存具体的数据
13 const state = {}
14
15 //创建并暴露store
16 export default new Vuex.Store({
17     actions,
18     mutations,
19     state
20 })
21
```

2. 在

main.js中创建vm时传入store配置项

```
1 .....
2 //引入store
3 import store from './store'
4 .....
5
6 //创建vm
7 new Vue({
8   el: '#app',
9   render: h => h(App),
10   store
11 })
12
```

```
11  })
```

```
12
```

3.4 基本使用

1. 初始化数据、配置

actions、配置mutations，操作文件store.js

```
1  //引入Vue核心库
2  import Vue from 'vue'
3  //引入Vuex
4  import Vuex from 'vuex'
5  //引用Vuex
6  Vue.use(Vuex)
7
8  const actions = {
9      //响应组件中加的动作
10     jia(context,value){
11         // console.log('actions中的jia被调用了',miniStore,value)
12         context.commit('JIA',value)
13     },
14 }
15
16 const mutations = {
17     //执行加
18     JIA(state,value){
19         // console.log('mutations中的JIA被调用了',state,value)
20         state.sum += value
21     }
22 }
23
24 //初始化数据
25 const state = {
26     sum:0
27 }
28
29 //创建并暴露store
30 export default new Vuex.Store({
31     actions,
```

```

32     mutations,
33     state,
34 })
35

```

2. 组件中读取vuex中的数据:

`$store.state.sum`

3. 组件中修改vuex中的数据:

`$store.dispatch('action中的方法名',数据)`或 `$store.commit('mutations中的方法名',数据)`

备注: 若没有网络请求或其他业务逻辑, 组件中也可以越过actions, 即不写dispatch, 直接编写commit 具体案例: index.js

```

1  //该文件用于创建Vuex中最为核心的store
2  import Vue from 'vue'
3  //引入Vuex
4  import Vuex from 'vuex'
5  //应用Vuex插件
6  Vue.use(Vuex)
7  //准备actions—用于响应组件中的动作
8  const actions = {
9    /* jia(context,value){
10      console.log('actions中的jia被调用了')
11      context.commit('JIA',value)
12    },
13    jian(context,value){
14      console.log('actions中的jian被调用了')
15      context.commit('JIAN',value)
16    }, */
17    jiaOdd(context,value){
18      console.log('actions中的jiaOdd被调用了')
19      if(context.state.sum % 2){
20        context.commit('JIA',value)
21      }
22    },
23    jiaWait(context,value){
24      console.log('actions中的jiaWait被调用了')
25      setTimeout(()=>{
26        context.commit('JIA',value)
27      },500)

```

```

28   }
29 }
30 //准备mutations—用于操作数据（state）
31 const mutations = {
32   JIA(state,value){
33     console.log('mutations中的JIA被调用了')
34     state.sum += value
35   },
36   JIAN(state,value){
37     console.log('mutations中的JIAN被调用了')
38     state.sum -= value
39   }
40 }
41 //准备state—用于存储数据
42 const state = {
43   sum:0 //当前的和
44 }
45 //创建并暴露store
46 export default new Vuex.Store({
47   actions,
48   mutations,
49   state,
50 })
51

```

Count.vue

```

1  <template>
2    <div>
3      <h1>当前求和为: {{$store.state.sum}}</h1>
4      <select v-model.number="n">
5        <option value="1">1</option>
6        <option value="2">2</option>
7        <option value="3">3</option>
8      </select>
9      <button @click="increment">+</button>
10     <button @click="decrement">-</button>
11     <button @click="incrementOdd">当前求和为奇数再加</button>
12     <button @click="incrementWait">等一等再加</button>

```

```
13 </div>
14 </template>
15 <script>
16     export default {
17         name: 'Count',
18         data() {
19             return {
20                 n: 1, //用户选择的数字
21             }
22         },
23         methods: {
24             increment(){
25                 // commit 是操作 mutations
26                 this.$store.commit('JIA',this.n)
27             },
28             decrement(){
29                 // commit 是操作 mutations
30                 this.$store.commit('JIAN',this.n)
31             },
32             incrementOdd(){
33                 // dispatch 是操作 actions
34                 this.$store.dispatch('jiaOdd',this.n)
35             },
36             incrementWait(){
37                 // dispatch 是操作 actions
38                 this.$store.dispatch('jiaWait',this.n)
39             },
40         },
41         mounted() {
42             console.log('Count',this)
43         },
44     }
45 </script>
46 <style lang="css">
47     button{
48         margin-left: 5px;
49     }
50 </style>
51
```

3.5 getters的使用

1. 概念：当state中的数据需要经过加工后再使用时，可以使用getters加工。

2. 在

store.js中追加getters配置

```
1  .....
2
3  const getters = {
4    bigSum(state){
5      return state.sum * 10
6    }
7  }
8
9  //创建并暴露store
10 export default new Vuex.Store({
11   .....
12     getters
13 })
14
```

3. 组件中读取数据：

`$store.getters.bigSum`

3.6 四个map方法的使用

导入

```
1  import {mapState, mapGetters, mapActions, mapMutations} from 'vuex'
2
```

1. **mapState方法：**用于帮助我们映射state中的数据为计算属性

```
1  computed: {
2    //借助mapState生成计算属性：sum、school、subject（对象写法）
3    ...mapState({sum: 'sum', school: 'school', subject: 'subject'}),
4
```

```
5 //借助mapState生成计算属性: sum、school、subject (数组写法)
6 ...mapState(['sum','school','subject']),
7 },
8
```

2. **mapGetters方法:** 用于帮助我们映射 getters 中的数据为计算属性

```
1 computed: {
2   //借助mapGetters生成计算属性: bigSum (对象写法)
3   ...mapGetters({bigSum:'bigSum'}),
4
5   //借助mapGetters生成计算属性: bigSum (数组写法)
6   ...mapGetters(['bigSum'])
7 },
8
```

3. **mapActions方法:** 用于帮助我们生成与 actions 对话的方法, 即: 包含 \$store.dispatch(xxx) 的函数

```
1 methods:{
2   //靠mapActions生成: incrementOdd、incrementWait (对象形式)
3   ...mapActions({incrementOdd:'jiaOdd',incrementWait:'jiaWait'})
4
5   //靠mapActions生成: incrementOdd、incrementWait (数组形式)
6   ...mapActions(['jiaOdd','jiaWait'])
7 }
8
```

4. **mapMutations方法:** 用于帮助我们生成与 mutations 对话的方法, 即: 包含 \$store.commit(xxx) 的函数

```
1 methods:{
2   //靠mapActions生成: increment、decrement (对象形式)
3   ...mapMutations({increment:'JIA',decrement:'JIAN'}),
4
5   //靠mapMutations生成: JIA、JIAN (对象形式)
6   ...mapMutations(['JIA','JIAN']),
7 }
```

备注：mapActions与mapMutations使用时，若需要传递参数需要：在模板中绑定事件时传递好参数，否则传的参数是事件对象(event)。 具体案例：

```

1  <template>
2    <div>
3      <h1>当前求和为: {{ sum }}</h1>
4      <h3>当前求和放大10倍为: {{ bigSum }}</h3>
5      <h3>年龄: {{ age }}</h3>
6      <h3>姓名: {{name}}</h3>
7      <select v-model.number="n">
8        <option value="1">1</option>
9        <option value="2">2</option>
10       <option value="3">3</option>
11     </select>
12     <!-- 用了mapActions 和 mapMutations 的话要主动传参 -->
13     <button @click="increment(n)">+</button>
14     <button @click="decrement(n)">-</button>
15     <button @click="incrementOdd(n)">当前求和为奇数再加</button>
16     <button @click="incrementWait(n)">等一等再加</button>
17   </div>
18 </template>
19 <script>
20 import { mapState, mapGetters, mapActions, mapMutations } from 'vuex'
21 export default {
22   name: "Count",
23   data() {
24     return {
25       n: 1, //用户选择的数字
26     };
27   },
28   computed: {
29     ...mapState(['sum', 'age', 'name']),
30     ...mapGetters(['bigSum'])
31   },
32   methods: {
33     ...mapActions({incrementOdd: 'sumOdd', incrementWait: 'sumWait'}),
34     ...mapMutations({increment: 'sum', decrement: 'reduce'})

```



```
35   },
36   mounted() {
37     console.log("Count", this);
38   },
39 };
40 </script>
41 <style lang="css">
42   button {
43     margin-left: 5px;
44   }
45 </style>
46
```

3.7 模块化+命名空间

1. 目的：让代码更好维护，让多种数据分类更加明确。

2. 修改

store.js

```
1  const countAbout = {
2    namespace:true,//开启命名空间
3    state:{x:1},
4    mutations: { ... },
5    actions: { ... },
6    getters: {
7      bigSum(state){
8        return state.sum * 10
9      }
10   }
11 }
12
13 const personAbout = {
14   namespace:true,//开启命名空间
15   state:{ ... },
16   mutations: { ... },
17   actions: { ... }
18 }
19
```

```
20 const store = new Vuex.Store({
21   modules: {
22     countAbout,
23     personAbout
24   }
25 })
26
```

3. 开启命名空间后，组件中读取state数据：

```
1 //方式一：自己直接读取
2 this.$store.state.personAbout.list
3 //方式二：借助mapState读取：
4 // 用 mapState 取 countAbout 中的state 必须加上 'countAbout'
5 ...mapState('countAbout', ['sum', 'school', 'subject']),
6
```

4. 开启命名空间后，组件中读取getters数据：

```
1 //方式一：自己直接读取
2 this.$store.getters['personAbout/firstPersonName']
3 //方式二：借助mapGetters读取：
4 ...mapGetters('countAbout', ['bigSum'])
5
```

5. 开启命名空间后，组件中调用dispatch

```
1 //方式一：自己直接dispatch
2 this.$store.dispatch('personAbout/addPersonWang', person)
3 //方式二：借助mapActions：
4 ...mapActions('countAbout', {incrementOdd: 'jiaOdd', incrementWait: 'jiaWait'})
5
```

6. 开启命名空间后，组件中调用commit

```
1 //方式一：自己直接commit
2 this.$store.commit('personAbout/ADD_PERSON',person)
3 //方式二：借助mapMutations:
4 ...mapMutations('countAbout',{increment:'JIA',decrement:'JIAN'}),
5
```

具体案例：

count.js

```
1 //求和相关的配置
2 export default {
3   namespaced:true,
4   actions:{
5     jiaOdd(context,value){
6       console.log('actions中的jiaOdd被调用了')
7       if(context.state.sum % 2){
8         context.commit('JIA',value)
9       }
10    },
11    jiaWait(context,value){
12      console.log('actions中的jiaWait被调用了')
13      setTimeout(()=>{
14        context.commit('JIA',value)
15      },500)
16    }
17  },
18  mutations:{
19    JIA(state,value){
20      console.log('mutations中的JIA被调用了')
21      state.sum += value
22    },
23    JIAN(state,value){
24      console.log('mutations中的JIAN被调用了')
25      state.sum -= value
26    },
27  },
28  state:{
29    sum:0, //当前的和
30    school:'尚硅谷',
```

```

31         subject: '前端',
32     },
33     getters: {
34         bigSum(state) {
35             return state.sum * 10
36         }
37     },
38 }
39

```

person.js

```

1  //人员管理相关的配置
2  import axios from 'axios'
3  import { nanoid } from 'nanoid'
4  export default {
5      namespaces: true,
6      actions: {
7          addPersonWang(context, value) {
8              if (value.name.indexOf('王') === 0) {
9                  context.commit('ADD_PERSON', value)
10             } else {
11                 alert('添加的人必须姓王! ')
12             }
13         },
14         addPersonServer(context) {
15             axios.get('
https://api.uixsj.cn/hitokoto/get?type=social
            axios.get('
16                 response => {
17                     context.commit('ADD_PERSON',
{id: nanoid(), name: response.data})
18                 },
19                 error => {
20                     alert(error.message)
21                 }
22             )
23         }
24     },
25     mutations: {

```

```

26         ADD_PERSON(state,value){
27             console.log('mutations中的ADD_PERSON被调用了')
28             state.personList.unshift(value)
29         }
30     },
31     state:{
32         personList:[
33             {id:'001',name:'张三'}
34         ]
35     },
36     getters:{
37         firstPersonName(state){
38             return state.personList[0].name
39         }
40     },
41 }
42

```

index.js

```

1  //该文件用于创建Vuex中最为核心的store
2  import Vue from 'vue'
3  //引入Vuex
4  import Vuex from 'vuex'
5  import countOptions from './count'
6  import personOptions from './person'
7  //应用Vuex插件
8  Vue.use(Vuex)
9  //创建并暴露store
10 export default new Vuex.Store({
11     modules:{
12         countAbout:countOptions,
13         personAbout:personOptions
14     }
15 })
16

```

count.vue

```

1 <template>
2   <div>
3     <h1>当前求和为: {{sum}}</h1>
4     <h3>当前求和放大10倍为: {{bigSum}}</h3>
5     <h3>我在{{school}}, 学习{{subject}}</h3>
6     <h3 style="color:red">Person组件的总人数是: {{personList.length}}</h3>
7     <select v-model.number="n">
8       <option value="1">1</option>
9       <option value="2">2</option>
10      <option value="3">3</option>
11    </select>
12    <button @click="increment(n)">+</button>
13    <button @click="decrement(n)">-</button>
14    <button @click="incrementOdd(n)">当前求和为奇数再加</button>
15    <button @click="incrementWait(n)">等一等再加</button>
16  </div>
17 </template>
18 <script>
19   import {mapState,mapGetters,mapMutations,mapActions} from 'vuex'
20   export default {
21     name: 'Count',
22     data() {
23       return {
24         n:1, //用户选择的数字
25       }
26     },
27     computed:{
28       //借助mapState生成计算属性，从state中读取数据。（数组写法）
29       ...mapState('countAbout',['sum','school','subject']),
30       ...mapState('personAbout',['personList']),
31       //借助mapGetters生成计算属性，从getters中读取数据。（数组写法）
32       ...mapGetters('countAbout',['bigSum'])
33     },
34     methods: {
35       //借助mapMutations生成对应的方法，方法中会调用commit去联系
36       mutations(对象写法)
37       ...mapMutations('countAbout',
38       {increment:'JIA',decrement:'JIAN'}),

```

```

37 //借助mapActions生成对应的方法，方法中会调用dispatch去联系
actions(对象写法)
38 ...mapActions('countAbout',
  {incrementOdd:'jiaOdd',incrementWait:'jiaWait'})
39 },
40 mounted() {
41   console.log(this.$store)
42 },
43 }
44 </script>
45 <style lang="css">
46   button{
47     margin-left: 5px;
48   }
49 </style>
50

```

person.vue

```

1  <template>
2    <div>
3      <h1>人员列表</h1>
4      <h3 style="color:red">Count组件求和为: {{sum}}</h3>
5      <h3>列表中第一个人的名字是: {{firstPersonName}}</h3>
6      <input type="text" placeholder="请输入名字" v-model="name">
7      <button @click="add">添加</button>
8      <button @click="addWang">添加一个姓王的人</button>
9      <button @click="addPersonServer">添加一个人，名字随机</button>
10     <ul>
11       <li v-for="p in personList" :key="p.id">{{p.name}}</li>
12     </ul>
13   </div>
14 </template>
15 <script>
16   import {nanoid} from 'nanoid'
17   export default {
18     name: 'Person',
19     data() {
20       return {
21         name: ''

```

```

22         }
23     },
24     computed:{
25         personList(){
26             return this.$store.state.personAbout.personList
27         },
28         sum(){
29             return this.$store.state.countAbout.sum
30         },
31         firstPersonName(){
32             return
33             this.$store.getters['personAbout/firstPersonName']
34         },
35         methods: {
36             add(){
37                 const personObj = {id:nanoid(),name:this.name}
38                 this.$store.commit('personAbout/ADD_PERSON',personObj)
39                 this.name = ''
40             },
41             addWang(){
42                 const personObj = {id:nanoid(),name:this.name}
43                 this.$store.dispatch('personAbout/addPersonWang',personObj)
44                 this.name = ''
45             },
46             addPersonServer(){
47                 this.$store.dispatch('personAbout/addPersonServer')
48             }
49         },
50     }
51 </script>
52

```

4. 路由

1. 理解： 一个路由（route）就是一组映射关系（key - value）， 多个路由需要路由器（router）进行管理。
2. 前端路由： key是路径， value是组件。

4.1 基本使用

1. 安装vue-router, 命令:

npm i vue-router

2. 应用插件:

Vue.use(VueRouter)

3. 编写router配置项:

```
1 //引入VueRouter
2 import VueRouter from 'vue-router'
3 //引入Luyou 组件
4 import About from '../components/About'
5 import Home from '../components/Home'
6
7 //创建router实例对象, 去管理一组一组的路由规则
8 const router = new VueRouter({
9   routes:[
10     {
11       path:'/about',
12       component:About
13     },
14     {
15       path:'/home',
16       component:Home
17     }
18   ]
19 })
20
21 //暴露router
22 export default router
23
```

4. 实现切换 (active-class可配置高亮样式)

```
1 <router-link active-class="active" to="/about">About</router-link>
2
```

5. 指定展示位置

```
1 <router-view></router-view>
2
```

4.2 几个注意点

1. 路由组件通常存放在pages文件夹，一般组件通常存放在components文件夹。
2. 通过切换，“隐藏”了的路由组件，默认是被销毁掉的，需要的时候再去挂载。
3. 每个组件都有自己的\$route属性，里面存储着自己的路由信息。
4. 整个应用只有一个router，可以通过组件的\$router属性获取到。

4.3 多级路由（多级路由）

1. 配置路由规则，使用children配置项：

```
1 routes:[
2   {
3     path: '/about',
4     component: About,
5   },
6   {
7     path: '/home',
8     component: Home,
9     children: [ //通过children配置子级路由
10      {
11        path: 'news', //此处一定不要写: /news
12        component: News
13      },
14      {
15        path: 'message', //此处一定不要写: /message
16        component: Message
17      }
18    ]
19   }
20 ]
```

```
17         }
18     ]
19 }
20 ]
21
```

2. 跳转（要写完整路径）：

```
1 <router-link to="/home/news">News</router-link>
2
```

3. 指定展示位置

```
1 <router-view></router-view>
2
```

4.4 路由的query参数

1. 传递参数

```
1 <!-- 跳转并携带query参数，to的字符串写法 -->
2 <router-link :to="/home/message/detail?id=666&title=你好">跳转</router-link>
3
4 <!-- 跳转并携带query参数，to的对象写法 -->
5 <router-link
6     :to="{
7         path: '/home/message/detail',
8         query: {
9             id: 666,
10            title: '你好'
11        }
12    }"
13 >跳转</router-link>
14
```

2. 接收参数：

```
1 $route.query.id
2 $route.query.title
3
```

4.5 命名路由

1. 作用：可以简化路由的跳转。

2. 如何使用

a. 给路由命名：

```
1 {
2   path: '/demo',
3   component: Demo,
4   children: [
5     {
6       path: 'test',
7       component: Test,
8       children: [
9         {
10          name: 'hello' //给路由命名
11          path: 'welcome',
12          component: Hello,
13        }
14      ]
15    }
16  ]
17 }
```

b. 简化跳转：

```
1 <!--简化前，需要写完整的路径-->
2 <router-link to="/demo/test/welcome">跳转</router-link>
```

```

3
4 <!--简化后，直接通过名字跳转 -->
5 <router-link :to="{name:'hello'}">跳转</router-link>
6
7 <!--简化写法配合传递参数 -->
8 <router-link
9     :to="{
10         name:'hello',
11         query:{
12             id:666,
13             title:'你好'
14         }
15     }"
16 >跳转</router-link>
17

```

4.6 路由的params参数

1. 配置路由，声明接收params参数

```

1 {
2   path: '/home',
3   component: Home,
4   children: [
5     {
6       path: 'news',
7       component: News
8     },
9     {
10      component: Message,
11      children: [
12        {
13          name: 'xiangqing',
14          path: 'detail/:id/:title', //使用占位符声明接收
15          component: Detail
16        }
17      ]
18    }
19  ]
20 }

```

```
18         }
19     ]
20 }
21
```

2. 传递参数

```
1  <!-- 跳转并携带params参数，to的字符串写法 -->
2  <router-link :to="/home/message/detail/666/你好">跳转</router-link>
3
4  <!-- 跳转并携带params参数，to的对象写法 -->
5  <router-link
6      :to="{
7          name:'xiangqing',
8          params:{
9              id:666,
10             title:'你好'
11         }
12     }"
13 >跳转</router-link>
14
```

特别注意：路由携带params参数时，若使用to的对象写法，则不能使用path配置项，必须使用name配置！

3. 接收参数：

```
1 $route.params.id
2 $route.params.title
3
```

4.7 路由的props配置

作用：让路由组件更方便的收到参数

```
1 {
2   name:'xiangqing',
```

```

3  path:'detail/:id',
4  component:Detail,
5  //第一种写法: props值为对象, 该对象中所有的key-value的组合最终都会通过props传给Detail组件
6  // props:{a:900}
7  //第二种写法: props值为布尔值, 布尔值为true, 则把路由收到的所有params参数通过props传给Detail
  组件
8  // props:true
9
10 //第三种写法: props值为函数, 该函数返回的对象中每一组key-value都会通过props传给Detail组件
11 props($route) {
12     return {
13         id: $route.query.id,
14         title:$route.query.title,
15         a: 1,
16         b: 'hello'
17     }
18 }
19 }
20

```

方便在要跳转去的组件里更简便的写法 跳转去组件的具体代码

```

1  <template>
2    <ul>
3      <h1>Detail</h1>
4      <li>消息编号: {{id}}</li>
5      <li>消息标题: {{title}}</li>
6      <li>a:{{a}}</li>
7      <li>b:{{b}}</li>
8    </ul>
9  </template>
10 <script>
11 export default {
12   name: 'Detail',
13   props: ['id', 'title', 'a', 'b'],
14   mounted () {
15     console.log(this.$route);
16   }
17 }
18 </script>

```

```
19 <style>
20 </style>
21
```

4.8 <router-link>的replace属性

1. 作用：控制路由跳转时操作浏览器历史记录的模式
2. 浏览器的历史记录有两种写入方式：分别为push和replace，push是追加历史记录，replace是替换当前记录。路由跳转时候默认为push
3. 如何开启replace模式：
<router-link replace>News</router-link>

4.9 程式路由导航

1. 作用：不借助<router-link> 实现路由跳转，让路由跳转更加灵活
2. 具体编码：

```
1 // $router 的两个API
2 this.$router.push({
3   name: 'xiangqing',
4   params: {
5     id: xxx,
6     title: xxx
7   }
8 })
9
10 this.$router.replace({
11   name: 'xiangqing',
12   params: {
13     id: xxx,
14     title: xxx
```



```
15         }
16     })
17     this.$router.forward() //前进
18     this.$router.back() //后退
19     this.$router.go() //可前进也可后退
20
```

4.10 缓存路由组件

1. 作用：让不展示的路由组件保持挂载，不被销毁。

2. 具体编码：

这个 include 指的是组件名

```
1 <keep-alive include="News">
2   <router-view></router-view>
3 </keep-alive>
4
```

4.11 两个新的生命周期钩子

作用：路由组件所独有的两个钩子，用于捕获路由组件的激活状态。

具体名字：

- activated路由组件被激活时触发。
- deactivated路由组件失活时触发。

这两个生命周期钩子需要配合前面的缓存路由组件使用（没有缓存路由组件不起效果）

4.12 路由守卫

1. 作用：对路由进行权限控制

2. 分类：全局守卫、独享守卫、组件内守卫

3. 全局守卫：

```
1 //全局前置守卫：初始化时执行、每次路由切换前执行
```

```

2 router.beforeEach((to,from,next)=>{
3     console.log('beforeEach',to,from)
4     if(to.meta.isAuth){ //判断当前路由是否需要权限控制
5         if(localStorage.getItem('school') === 'zhejiang'){ //权限控制的具体规则
6             next() //放行
7         }else{
8             alert('暂无权限查看')
9             // next({name:'guanyu'})
10        }
11    }else{
12        next() //放行
13    }
14 })
15
16 //全局后置守卫：初始化时执行、每次路由切换后执行
17 router.afterEach((to,from)=>{
18     console.log('afterEach',to,from)
19     if(to.meta.title){
20         document.title = to.meta.title //修改网页的title
21     }else{
22         document.title = 'vue_test'
23     }
24 })
25

```

完整代码

```

1 // 这个文件专门用于创建整个应用的路由器
2 import VueRouter from 'vue-router'
3 // 引入组件
4 import About from '../pages/About.vue'
5 import Home from '../pages/Home.vue'
6 import Message from '../pages/Message.vue'
7 import News from '../pages/News.vue'
8 import Detail from '../pages/Detail.vue'
9 // 创建并暴露一个路由器
10 const router = new VueRouter({
11     routes: [
12         {

```

```

13     path: '/home',
14     component: Home,
15     meta:{title:'主页'},
16     children: [
17         {
18             path: 'news',
19             component: News,
20             meta:{isAuth:true,title:'新闻'}
21         },
22         {
23             path: 'message',
24             name: 'mess',
25             component: Message,
26             meta:{isAuth:true,title:'消息'},
27             children: [
28                 {
29                     path: 'detail/:id/:title',
30                     name: 'xiangqing',
31                     component: Detail,
32                     meta:{isAuth:true,title:'详情'},
33                     props($route) {
34                         return {
35                             id: $route.query.id,
36                             title:$route.query.title,
37                             a: 1,
38                             b: 'hello'
39                         }
40                     }
41                 }
42             ]
43         }
44     ]
45 },
46 {
47     path: '/about',
48     component: About,
49     meta:{ title: '关于' }
50 }
51 ]
52 })

```

```

53 // 全局前置路由守卫——初始化的时候被调用、每次路由切换之前被调用
54 router.beforeEach((to, from, next) => {
55     console.log('前置路由守卫', to, from);
56     if(to.meta.isAuth) {
57         if(localStorage.getItem('school') === 'zhejiang') {
58             // 放行
59             next()
60         } else {
61             alert('学校名不对，无权查看')
62         }
63     } else {
64         next()
65     }
66 })
67 // 全局后置路由守卫——初始化的时候被调用、每次路由切换之后被调用
68 router.afterEach((to, from) => {
69     console.log('后置路由守卫', to, from)
70     document.title = to.meta.title || '我的系统'
71 })
72 export default router
73

```

1. 独享守卫:

就是在 routes 子路由内写守卫

```

1  beforeEnter(to,from,next){
2      console.log('beforeEnter',to,from)
3      if(to.meta.isAuth){ //判断当前路由是否需要权限控制
4          if(localStorage.getItem('school') === 'atguigu'){
5              next()
6          }else{
7              alert('暂无权限查看')
8              // next({name:'guanyu'})
9          }
10     }else{
11         next()
12     }
13 }
14

```

5. 组件内守卫:

```
//创建并暴露一个路由器
const router = new VueRouter({
  routes:[
    {
      name:'guanyu',
      path:'/about',
      component:About,
      meta:{title:'关于'}
    },
    {
      name:'zhuye',
      path:'/home',
      component:Home,
      meta:{title:'主页'},
      children:[
        {
          name:'xinwen',
          path:'news',
          component:News,
          meta:{isAuth:true,title:'新闻'},
          beforeEnter: (to, from, next) => {
            console.log('独享路由守卫',to,from)
            if(to.meta.isAuth){ //判断是否需要鉴权
              if(localStorage.getItem('school')==='atguo')
                next()
            }else{
              alert('学校名不对，无权限查看！')
            }
          }else{
            next()
          }
        }
      ]
    }
  ]
})
```

CSDN @格雷狐思

在具体组件内写守卫

```
1 //进入守卫：通过路由规则，进入该组件时被调用
2 beforeRouteEnter (to, from, next) {
3 },
4 //离开守卫：通过路由规则，离开该组件时被调用
5 beforeRouteLeave (to, from, next) {
6 }
7
```

vue_test > 42_src_组件内路由守卫 > pages > ▼ About.vue > {} "About.vue" >  template

```

1  <template>
2    <h2>我是About的内容</h2>
3  </template>
4
5  <script>
6    export default {
7      name: 'About',
8      /* beforeDestroy() {
9        console.log('About组件即将被销毁了')
10      }, */
11      /* mounted() {
12        console.log('About组件挂载完毕了', this)
13        window.aboutRoute = this.$route
14        window.aboutRouter = this.$router
15      }, */
16      mounted() {
17        // console.log('%%%', this.$route)
18      },
19
20      //通过路由规则，进入该组件时被调用
21      beforeRouteEnter (to, from, next) {
22        console.log('About--beforeRouteEnter', to, from)
23        if(to.meta.isAuth){ //判断是否需要鉴权
24          if(localStorage.getItem('school')=== 'atguigu'){
25            next()
26          }else{
27            alert('学校名不对，无权限查看！')
28          }
29        }else{
30          next()
31        }
32      },
33
34      //通过路由规则，离开该组件时被调用
35      beforeRouteLeave (to, from, next) {
36        console.log('About--beforeRouteLeave', to, from)
37        next()
38      }
39    }
40  </script>

```

4.13 路由器的两种工作模式

1. 对于一个url来说，什么是hash值？—— #及其后面的内容就是hash值。

2. hash值不会包含在 HTTP 请求中，即：hash值不会带给服务器。

3. hash模式：

- a. 地址中永远带着#号，不美观。
- b. 若以后将地址通过第三方手机app分享，若app校验严格，则地址会被标记为不合法。
- c. 兼容性较好。

4. history模式：

- a. 地址干净，美观。
- b. 兼容性和hash模式相比略差。
- c. 应用部署上线时需要后端人员支持，解决刷新页面服务端404的问题。

参考文章：



```
import News from '../pages/News'
import Message from '../pages/Message'
import Detail from '../pages/Detail'

//创建并暴露一个路由器
const router = new VueRouter({
  mode: 'history',
  routes: [
    {
      name: 'guanyu',
      path: '/about',
      component: About,
      meta: {isAuth: true, title: '关于'}
    },
    {
      name: 'zhuye',
      path: '/home',
      component: Home,
      meta: {title: '主页'}
```

- 1. 尚硅谷主讲的vue: https://www.bilibili.com/video/BV1Zy4y1K7SH?p=77&spm_id_from=pageDriver
- 2. 现代JavaScript: <https://zh.javascript.info/>
- 3. MDN文档: <https://developer.mozilla.org/zh-CN/docs/Web>
- 4. 我是你的超级英雄 <https://juejin.cn/post/6844903895467032589>